A Sheet Music Organisation System

Interim Report

Submitted for the BSc in Computer Science with Industrial Experience

January 2015

by

Charlotte Godley

Contents

1 Introduction

This project focuses on the organisation and display of sheet music. Sheet music refers only to the instructions given to a performer in order to play a composition, and does not include the sound output produced when the piece is performed. Any reference to music from this point onward should be assumed to mean visual sheet music, rather than audio recordings. Whilst Eastern countries and previous eras have used different methods of notation (Kaufman, 1967), this project focuses solely on the notation used by western classical music. This type of musical notation is the format most commonly used by orchestras and performers, and in order to understand the problem of organising it, some explanation of the key elements will be required.

Notation of western classical music has used a combination of diastematic and orthographic notation (Rastall, 1982) since the advent of Gregorian chant around 640AD (Taruskin, 2005). The function of this notation falls into two main divisions: the expression of relationship in sound frequency, and the expression of relationship in time, or measure (Warner, 1929).

The representations of other parameters in staff notation are normally orthographic, such as indications p - meaning piano, or "quiet" - and pizz - meaning pizzicato, or "plucked" (Rastall, 1982). This mechanism is complex in nature, and has a large but finite set of symbols which control every element of the composition. This will be discussed and explained later in this report.

Despite the advantages of digitising music collections notated using this system, the majority of musicians choose to store music physically, using filing cabinets and music cabinets as storage mechanisms(Sullivan, 2012). This can be attributed to the lack of standardisation for browsing and organising digital sheet music. PDF is the standard digitisation method for documents, which presents a problem for musicians wanting to search by multiple methods because the format does not include meta information about what the document contains,(Good, 2013) as well as the usability of PDF browsers in musical performances. Whilst the latter has largely been solved by tablet applications(ForScore, LLC, 2014), organisation, searching and filtering PDF files is still largely a manual task, with most applications allowing for manual input, but with little to none handling automatic information retrieval(MusicReader, 2015). Further alternative software solutions proving this point will be discussed later in the report.

A further problem in digitising music collections is merging online collections with offline collections, which is not improved by many online and offline music retailers providing only physical copies of compositions(Musicroom, 2015). In other words, as collections of music grow, it becomes a long and arduous task to convert physical documents to digital files, and from there to manually attribute meta data for each and every piece.

The physical method avoids this problem, but is still fraught with problems of manual organisation, made more difficult by the need to index or else duplicate files in order to organise collections by multiple methods. This problem is due to music cabinets only allowing the user to view the top document, as well as sheet music books often only providing bibliography information on the front cover, meaning the spine of books does not tell a user what the book contains(Feist, 2012).

This is further complicated by specific use cases, such as music cabinets used by multiple musicians who each play several different instruments. In a solo use case, it may be possible to organise by title, instrument, or composer, but in the given use case finding one which unifies all users needs is difficult.

A further example would be a musical director for an orchestra, who wishes to not only find a specific piece, but find compositions which would work well in a concert schedule. In this case it would be of use to know more than the bibliography of a piece, but information such as time, speed and instruments information without having to physically look or memorise each piece's information. For this reason many larger Orchestras have a dedicated Orchestral Librarian(Association, 2015), who will handle manual organisation, maintenance and research of the library. This indicates that library sizes take a lot of maintenance, and manual conversion and extraction of meta data to convert a physical library to a digital collection would take a long time.

As such, this project solves the organisation problem by extracting meta data about sheet music automatically, with additional features provided in order to improve the usability and shorten the amount of time needed to create and expand a digital collection of music. This document discusses the aims and objectives of this project, technical scope and depth of the project, process and method used to produce the solution, and finally critical evaluation of the project as a whole.

2 Aims and Objectives

2.1 Project Aim

The overall aim of the project is to design and develop a sheet music library application, with the ability to organise and view personal sheet music collections, and download sheet music from the internet. Time permitting, it should also be able to generate sound from the sheet music, and import editable music from flat images.

2.2 Primary Objectives

The following objectives are of the highest importance to the project, and are a measure of whether the project has been completed.

- **Rendering of Musical Files**
  It will be necessary to render one or more formats of commonly stored musical files, as the aim of the project is to enable users to view and organise their sheet music collections.
- **Extraction of Metadata**
  The project will be required to extract important information from each piece, ranging from the simple nominal data such as title, composer, lyricist, to the more complex notation such as clefs, key signatures and meters used.
- **Ability to search metadata extracted and auto-generate playlists**
  From the extracted metadata it should be possible to search the catalog of music for specific requirements, such as key, clef, meter, time signature (explained in the background section). The system should also be capable of generating playlists based on related data.
- **Connection to Online Music Collections**
  It should also be possible to connect to online music collections, as it would be beneficial to users to be able to search and add to their collections using the same interface.

2.3 Secondary Objectives

The secondary objectives are to be completed only if they do not threaten the completion of primary objectives.

- **Audio playback**
  It would be useful to a cross section of users to be able to play music files as sound clips. This enables performers who regularly play with an accompany musician or ensemble to create practice accompaniments from their sheet music, or hear an approximation of how a melody should sound.
- **MusicOCR conversion of images to parseable Music files**
  It would be easier for musicians to merge their physical and virtual music collections for automatic organisation if the solution provided a way to import flat, scanned sheet music into marked-up music files.

3 Background

3.1 Problem Context

This problem's main focus is the difficulty of organising classical sheet music, and how this can be made easier by the automatic extraction of key pieces of information. In order to understand what a performer may want to know about a particular piece, it is important to have a brief understanding of the elements of musical notation common to all compositions.

The key element of this form of notation is the staff, as shown in figure 1. This is a grouping of five horizontal lines, with each line or space in the staff indicating a different sound pitch, a term meaning the relative "highness" or "lowness" of the sound (Land and Vaughan, 1978).

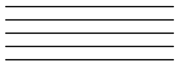


Figure 1: A blank staff

This staff is divided by bar lines, vertical lines delineating grouped units of sound and silence (formally referred to as notes and rests), which provides an indication of the unit's relationship in time by its juxtaposition to other groupings in the composition. These groupings are called measures or bars, with each bar having a variable maximum of notes and rests.

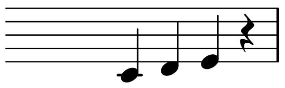


Figure 2: A bar containing three notes and one rest

3.1.1 Clefs

In the system of staff notation, sound frequencies, or pitches, are denoted by letters A-G - after each cycle of the letter names, the next pitch above it will be the start of a new cycle. The cycles are often split by octaves, a term meaning eight pitches, for example A to A or E to E.

In order to provide a link between the lines and spaces of a staff and pitch name, a clef symbol is necessary.

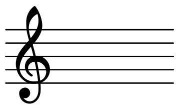


Figure 3: A staff with a treble clef

Each clef symbol denotes a different pitch name - figure 3 shows a G. The center around which this symbol is drawn - in figure 3, the second line from the bottom of the staff - indicates that this line or space will be known as the pitch name denoted by the symbol. From this the reader can infer all other pitches by counting through the letters of the cyclic octave system, so in figure 3, the pitch above becomes an A, and the pitch below becomes an F.

This symbol is important to a musician as different clefs are used to position the majority of the pitches in a piece on the staff, as this makes it easier to read. From this a performer can infer the average range of a piece, and predict whether this will be comfortable for the performer's chosen instrument or voice.

3.1.2 Keys

A second important indication to the player is the key, denoted by a key signature.

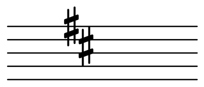


Figure 4: A staff with a key signature

A collection of symbols at the beginning of the piece indicate which pitches should be raised by half pitches, and which should be lowered. Raised pitches are called sharps, indicated by the # symbol, whilst lowered pitches are called flats, indicated by the ♭ symbol. Each key, which has a letter name and key type (which can either be "major" or "minor"), has a different combination of flats or sharps.

This is a useful piece of notation to a musician as pieces in less common keys, such as C# major or F# major, may prove more difficult for the user to perform, and therefore they may want to filter out pieces in these particular keys. Similarly, in the case of singers, a singer's range may sit comfortably in one or two keys and they would perhaps want to find pieces in only these keys.

3.1.3 Meter

The third symbol denoted at the beginning of a measure is the meter or time signature, displayed as two numerals positioned like a mathematical fraction.

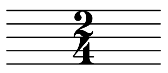


Figure 5: A staff with a 2/4 time signature, or meter

The upper number of a meter symbol indicates the amount of beats in the bar. A beat simply refers to a note or rest, and the type of beat is indicated by the lower number. In the case of figure 5, 2/4 indicates a measure will contain 2 crotchets, or quarter-length notes. The most common time signature is 4/4, which for this reason is usually denoted with a C in place of the fraction, meaning "Common time".

This information is important as it tells the performer how the rhythm and beat of the piece should be felt, counted and performed, and is useful for searching purposes as different meters give the piece a different feeling, dictating the sort of occasion this piece would accompany.

For example, 2/4 is commonly used for march pieces, and 3/4 is commonly used for waltzes and dance pieces.

3.1.4 Tempo

The speed of a particular piece, or the tempo, is indicated by an equation.

The equation above the staff in figure 6 indicates that the piece should be played at 60 beats per minute. The symbol dictating the sort of beat per minute depends on the time signature, here a crotchet (or quarter note)

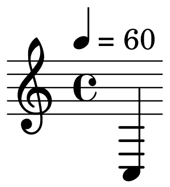


Figure 6: A staff with tempo marking

is given as the piece is in 4/4 time. Sometimes, this will be accompanied by a text direction to indicate speed or style, such as Andante, indicating a walking speed.

This indication would prove a useful identifier as pieces of different tempos provide variation in performance lists, so a concert organiser may want to find pieces with a variety of tempos.

3.1.5 Further metadata

Aside from these symbols, there are some items of textual information useful to the user.

The first of these would be the parts in the piece and their transpositions. A part refers to a grouping of measures given to one performer, as shown in figure 7. "Part" used in a general sense usually refers to the names given to the left, in this example, "Clarinet in B♭" and "Flute".

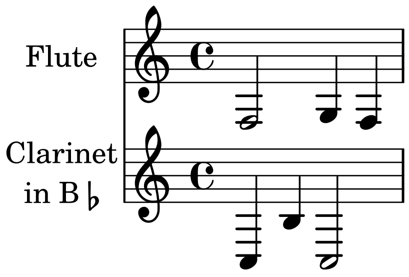


Figure 7: Two separate parts in one score

Parts would be relevant as a particular group of instrumentalists may need parts that fit their instruments. If this is not the case for a given piece, however, a part written for a different instrument, for example, the Alto Saxophone rather than the Tenor Horn, may be compatible with the instrument anyway, if the transposition matches the instruments together.

An instrument which has a transposition means that, whilst most instruments would play a note as it is written, a transposing instrument will automatically sound the note in a different key, as described earlier, which may raise or lower the sound of the instrument. For example, the note C played on an Alto Saxophone will sound as an E♭, because it is in the key of E♭ major.

Further to this, the user would want to know the piece's title, and names of publishers, composers, arrangers and lyricists of the work. Further to the composer name, it may be useful to know the date of composition as an indication of the era in which the piece was composed, such as Classical/Baroque/Romantic, though this would not always be written on the sheet music so may need to be researched using the internet.

3.2 Comparison of Technologies

3.2.1 Programming Language

This project could be developed with a variety of programming languages, as displayed in table 1.

Language	Speed of development	Developer's Knowledge	Most recent use
C#	Fast	A lot	2nd year
Python	Fast	A lot	In constant use for over a year
C++	Slow	Average	2nd year

Table 1: Table of languages considered

The three key elements of whether a language is suitable for this project are speed of development, as the time constraint of a year means it is important that development is not hindered by the language itself, developer knowledge and most recent usage of the language as this will provide an additional time benefit. These are displayed in table 1.

A further consideration is platform independence, as the developer intends to make the project accessible to all users. It is understood that C# is platform independent through the use of the Mono Project, which is feature-complete to C# 10 (The Mono Project, 2015), or Xamarin Studio and other such tools, but the developer has not developed any applications with C# for use on multiple operating systems. For this reason, the developer feels more comfortable using Python, owing to the experience of writing applications for Linux and Windows in previous projects.

The developer has also considered the Open Source communities and repositories around each of the languages. Whilst not important in the context of the current project, after the project is completed it is intended that the project be Open Sourced to contribute to further research in this area, and considering the graph in figure 8 (Redmonk, 2014), Python has the highest percentage of repositories on the popular Open Source repository website Github of the three languages considered. Whilst this does not categorically prove it as the best language for Open Source software development, it shows that there are a high number of users and projects in this area, which will make it a more popular project to work with once open sourced.

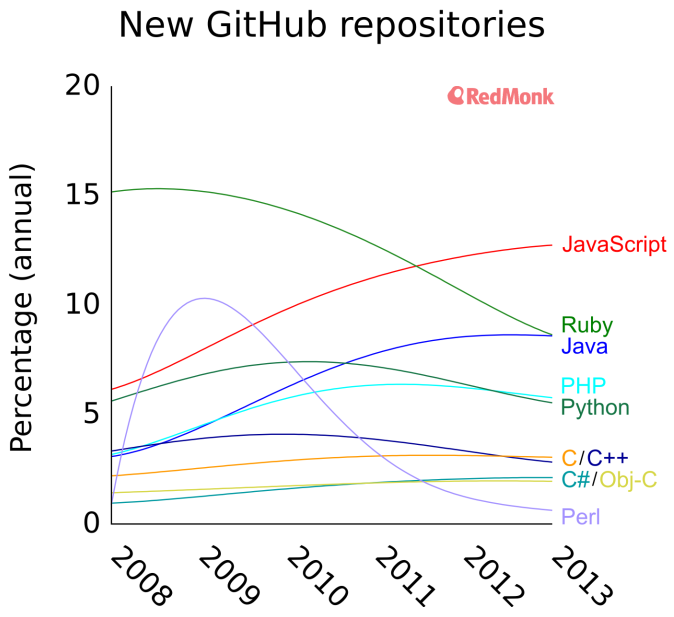


Figure 8: A graph showing the percentage of repositories on Github in different languages

Furthermore, there are many projects in the field of musical software research currently in existence using this language, (The Python Foundation, 2015) such as mingus, an advanced music theory and notation package for Python.

It is for these reasons that Python has been chosen as the development language. Beyond the selection of Python, it is important to discuss and consider which version to use, as Python 3 was introduced in 2008, but Python 2 continues to be maintained and was updated to include many of the backwards compatible features in 2010. Whilst this seems an obvious choice as Python 3 is the latest version, many projects still have issues updating and using Python 3 as it was deliberately not backwards compatible (The Python Foundation, 2012).

Upon due consideration, Python 3 has been selected, though if the project requires legacy libraries this may have to be evaluated again. This decision has been made because at this stage the project does not appear to require libraries which do not work with Python 3, and therefore the developer should make every effort to keep the project up to date with the latest version.

3.2.2 File format

The project will require at least one default format for it to process music, which needs to have detailed information about what the score contains. Table 2 describes the options considered.

Format	Purpose
muscx	MuseScore notation
SIB	Sibelius notation
new format	this project only
MusicXML	sharing music between software

Table 2: A table showing the different file formats considered

The first two options, muscx and SIB files, are formats used by the open source notation software MuseScore (MuseScore, 2015a), and the world's most popular proprietary notation software, Sibelius (Avid, 2015a). Using either or both of these files would mean the majority of users would be able to use the application. However, both options couple this project with those particular packages, when users could still choose other software to write music with. Furthermore, the formats are specifically designed for those software packages and may have nuances which make development for this project more difficult. Additionally, Sibelius is proprietary so borrowing their file format may cause copyright issues.

The third option is to create an entirely new format. This would mean the file format was designed to the requirements of the project and therefore be entirely customisable and extensible. However, this project is created with the intention of organising, not composing music, so the files would have to be created or imported from other software packages manually if the project does not include composition. Therefore, this is the least applicable option.

The fourth and final option is MusicXML, a file format intended for sharing and archiving the world's sheet music (Make Music, Inc, 2015). This particular format is used by a wide variety of software packages (Make Music, Inc, 2015) and is included in the formats usable by both MuseScore (MuseScore, 2015a) and Sibelius (Avid, 2015a), therefore neither couples the format with a program nor requires manual creation and import of current music files. However, this particular format was designed by a third party, and might therefore present a further technical challenge in learning how the format notates everything, which will have been designed according to the requirements of the original developer, Make Music (Make Music, Inc, 2015), and may not reflect the same design intentions as this project.

It would also be possible to create or include file format translators to MusicXML in the project. MusicXML has been the most successful at standardising music file formats (Make Music, Inc, 2012), and therefore there are a multitude of projects which have translated various popular formats into MusicXML. For this reason, the project's default format will be MusicXML.

3.3 Comparison of Algorithms for Rendering and Organising Sheet Music

Figure 9 shows a flow diagram for the system of rendering and organising sheet music. Where there are process markings which are specific to this project,(unzipping is not counted in this category), these will be described and analysed in the following sections. Mxl files are the format used to compress MusicXML files, and as such the procedure for decompressing them is not complicated. Where arrows have text, this indicates that the arrow will only be followed if the text condition is met.

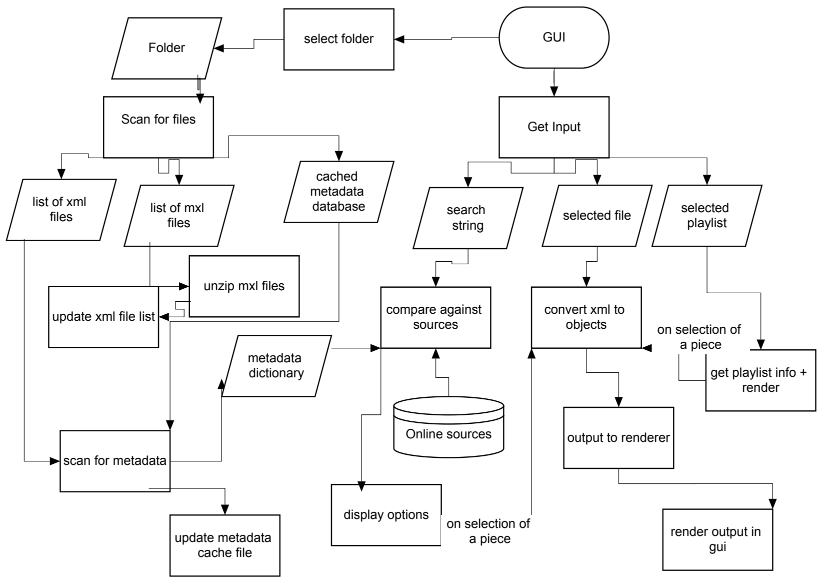


Figure 9: A flow diagram describing the rendering and organising system

3.3.1 Algorithms for parsing XML to Objects

For the rendering of sheet music, it will be necessary to parse a musicXML file into a hierarchy of objects, beginning with the overall piece and descending into each part and measure.

For the parsing of XML itself, there are two potential built in methods to choose from. The first, known as DOM or Document Object Model, loads the entire XML file into memory and provides methods to search the loaded file for specified tags. The developer has used this before in personal and industrial projects, and believes it is cumbersome to manipulate data in this way. Furthermore, this project is focussing on rendering the information rather than rendering it with precise formatting, and many software packages implant musicXML files with very complex formatting information which may or may not be necessary.

The second option is using a different api called the Simple API for XML (SAX). In this method, the program loads the XML file tag by tag, and connects to call backs when specified things occur in the file, for example a new tag or piece of data inside tags, or the closing of an old tag. This is easier to work with as functionality can iteratively be built up by creating handlers for each tag, and is better for memory management as only tags which are necessary to the project will have any effect on the object structure. For these reasons, this method has been selected.

3.3.2 XML verification algorithm

For both the algorithm options discussed in section 3.3.1, a further choice is whether to verify the XML parsed, using an online file validator, or presume the file is written in valid MusicXML.

The usual choice is to verify all XML, and is therefore the default option for both methods of parsing. Whilst this confirms that XML is valid before starting parsing of a file which could be corrupt, the speed at which files will parse is greatly reduced according to the speed of the user's internet connection. Furthermore, if the user is browsing their own music collection, it should not be necessary for the user to be connected to the internet.

Due to speed and functionality considerations, the choice has been made that the XML parser algorithm will not verify XML being converted to objects, or being examined for metadata. Given that most musicXML will be produced automatically by other programs, it is unlikely files opened by the project will be corrupt, though necessary steps will be taken to avoid this causing a problem in the program.

3.3.3 Metadata Scanning algorithm

The metadata algorithm has been designed so that, for a given folder, the program will parse all of the files with the XML extension for a given selection of information (for example, composer, piece title, instruments). Based on development and testing of this algorithm, the data will be cached to a permanent file in order to ensure unchanged files do not have to be parsed more than once.

This is to be indexed either by the information title - e.g "composer"; or by the information itself - e.g "bartok". This will facilitate faster searching of the database for use when the user is finding a particular piece, and simplify the production of auto generated playlists by the system. Depending on the method of indexing, the alternate indexer should be stored as part of the value in a key value pair format, alongside the file in which it was found.

It has been decided that in memory, this will be structured using a generic type. This decision has been taken because only 3 pieces of information per item of data will be stored, one of which will be the index of the data, so using a dictionary holding tuples would be more appropriate than creating an object as it simplifies the algorithm and therefore the debugging process.

Further to this, of the two xml algorithms described in section 3.3.1, the algorithm will use SAX, as the described metadata algorithm does not require all of the data and would therefore waste a lot of memory by loading in the entire file.

Finally, the initial implementation of this algorithm used a serialised copy of the generic type to store the metadata to file. It has been decided for the purposes of extensibility and portability that data will be stored to an SQLite file, a light implementation of SQL databases. This is a more complex file type and as such will take longer to implement, but ensures that if this project is developed on in the future, that the file format can be used by other platforms and languages without converting to and from a Python object.

3.3.4 Rendering Algorithm

The program is required to take the object structure and transform it, in some way, to musician readable sheet music. The user should be able to pan around the sheet music and zoom in and out of it to view specific details.

This could be achieved using an entirely new algorithm, with the output going directly to the render window using different glyphs and fonts extracted from their relevant classes.

However, the functionality of panning and zooming using this algorithm may be difficult to optimise, as both could possibly require running the algorithm each time the user provides input.

Furthermore, the conversion of even basic sheet music to a readable format would require a high level of precision and complexity, and creating a new algorithm would be considered reinventing the wheel, so to speak, as this is a process that has been covered by many different applications (like MuseScore (MuseScore, 2015a), Finale (Make Music, Inc, 2015) and Sibelius (Avid, 2015a)). Lastly, the process of debugging whether the symbols are correct would require visual checking and would be difficult to debug automatically.

It would be possible to alleviate the panning and zooming problem by converting the collection of symbols to an image or PDF file and using a built in image rendering library, such as wxPython (The wxPython Team, 2012). However, this method still involves reinventing the wheel and problems with visual debugging being required.

Considering these factors, it has been decided that the algorithm will be outputting files to a third party system known as Lilypond. Lilypond is a language and system developed to typeset the highest quality sheet music (Lilypond, 2015), which takes an input file and outputs a PDF or image. As this is a language unto itself and has been in development for many years by the Open Source community, this will alleviate the problem of visual debugging - instead, each class can create a formatted Lilypond output based on its attributes and unit tests can automatically confirm that the result is as expected.

3.4 Comparison of Technologies for Importing Online Musical Sources

3.4.1 Musical Sources

Two open and free sources of sheet music have been selected for potential inclusion, which will enable users to increase their own music collections with new music without using a browser to peruse collections. The first is **MuseScore Online**, which is a community website created for composers to upload, share and discover compositions using the MuseScore platform (MuseScore, 2015b).

This has been selected due to the number of files available, the openness of the platform and the well documented API. It will, however, be necessary to manage copyright issues, as pieces published on this website may be published under the license of the composer's choosing and therefore may cause issues with certain types of users, in particular those performing commercially.

The second selected source is the **IMSLP**. This is the **International Music Score Library Project**, built with the intention of sharing the world's public domain music, and contains 290,000 scores to date (IMSLP, 2015). This may be a questionable source, as not all pieces are uploaded in MusicXML format due to the pieces being scanned and uploaded by community members, rather than being automatically generated by a piece of software. However, this source does not raise any copyright issues as all pieces are no longer covered by copyright.

It may also be possible to import collections from subscription services and websites enabling purchase of music, such as **MusicNotes.com**. However, this will require closer contact with the companies maintaining the website and may not be appropriate for an educational and academic purpose.

3.4.2 Searching Algorithm

The APIs for both selected sources provide a variety of output formats, the 2 most prominent being XML and JSON. The algorithm for browsing the source from the program will need to in some way, contact the server to confirm whether there are pieces which have a specific attribute entered by the user, and download the file if the piece is selected.

It would be possible to use an algorithm which repeatedly connects to the API and polls for the relevant input from the user, returning a list of options which the user would then select from and download from the server. Whilst this would be simple to implement, this would make the program considerably slower, as it would require repeated connection to the internet. This would also cause problems for the maintainers of the server, as repeated requests from a piece of software would cause a heavy load on the server.

If possible, the software should cache a copy of all metadata served from each online source to alleviate this problem, and search for the relevant inputted data from this, and then, if necessary, collect the relevant file from the server. This would require a connection to the server only twice - once when updating metadata sources, and once when downloading a file - rather than a persistent or repeated connection.

However, this needs confirmation from the sources selected that caching is an accepted API algorithm, as some websites stipulate that this is not allowed, and therefore the present decision is to use the previous option until this has been confirmed.

3.5 Comparison of Algorithms for Sound Output and Image Input

3.5.1 MIDI algorithm

The sound output algorithm must, for a given part or selection of parts, output the sheet music to a MIDI or MP3 file, which can then be played within the program.

It has been decided that each class in the solution will have a method to produce this output, in the same way as the algorithm described for rendering in section 3.3.4, which will be combined into an output file and played.

This creates an extendible architecture, as it would easily be possible to create output methods to other formats in the future.

3.5.2 Image input algorithm

In order to import images or flat files into the chosen file format, it will be necessary for the program to include the ability to apply music optical character recognition to the file, and save the output to MusicXML, which can then be parsed by other parts of the program.

It would be possible for a new algorithm to be produced for converting new imported images into the chosen file format. This would mean the algorithm could be optimised according to the project aims, and provide sufficient technical challenge.

However, this project is concerned with music organisation, not optical music recognition specifically, and as such the project is too large to commit a sufficient amount of time to this particular algorithm in order to make it function as well as other algorithms.

As a reference point, Optical Character Recognition for natural languages has taken many years to develop and perfect, and has been an attractive research area and idea to a wide variety of users (Fujisaki et al., 1990). OMR, or Optical Music Recognition, has been the focus of international research for over three decades, and while numerous achievements have been made, there are still many challenges to be faced before it reaches its full potential (Bainbridge and Bell, 2015).

It has therefore been decided that OCR as a topic is too large for this project, and if this goal is included in the project, it will be through communication with other systems, such as Audiveris, an open music scanner (Audiveris, 2015).

This removes the technical challenge of producing an entirely new algorithm, but adds the challenge of understanding how optical music recognition scanners work, and how they can be integrated with the system, particularly if the third party package is not developed in Python.

3.6 Alternative Solutions

Table 3 shows the alternative options considered in the area of Sheet Music organisation automation. This shows that the closest alternative would be Power Music Pro, though much of the functionality changes slightly according to the platform it has been developed for (SightRead Ltd., 2015). Furthermore, Power Music Pro's only improvement on manual organisation is the ability to search by lyric, whilst this project intends to allow for a cross section of other organisation techniques, as explained in the problem context.

Additionally, each of the possible options are released in a commercial environment, with Avid's Photoscore being too expensive for the average user. Seemingly, this project would constitute the only free and Open Source software released for this problem.

A final point to make is that none of these solutions provide a version for Linux based operating systems, whilst this project should be useable on Mac, PC and Linux based operating systems.

Software	Rendering of Sheet Music	Manual Organisation	Automatic Organisation by complex notation	Connection to Online Sources	Audio Playback	OMR	Price	Platform
Avid Scorch	✓	✓	×	✓	✓	×	£1.40 (Avid, 2013)	iOS
Power Music Pro/Power Music Mac	✓	✓	partial	✓	✓	×	£49 for PC, £29 for Mac (SightRead Ltd., 2015)	PC & Mac
Avid Photoscore	✓	×	×	×	×	✓	£200 (Avid, 2015b)	PC & Mac
Scorcerer	✓	✓	×	×	✓	✓	£15 for iPad version, £26 for Mac and PC (Deskew Technologies, 2011)	iPad, Mac & PC

Table 3: A comparison table of other available software

4 Designs

4.1 System Design

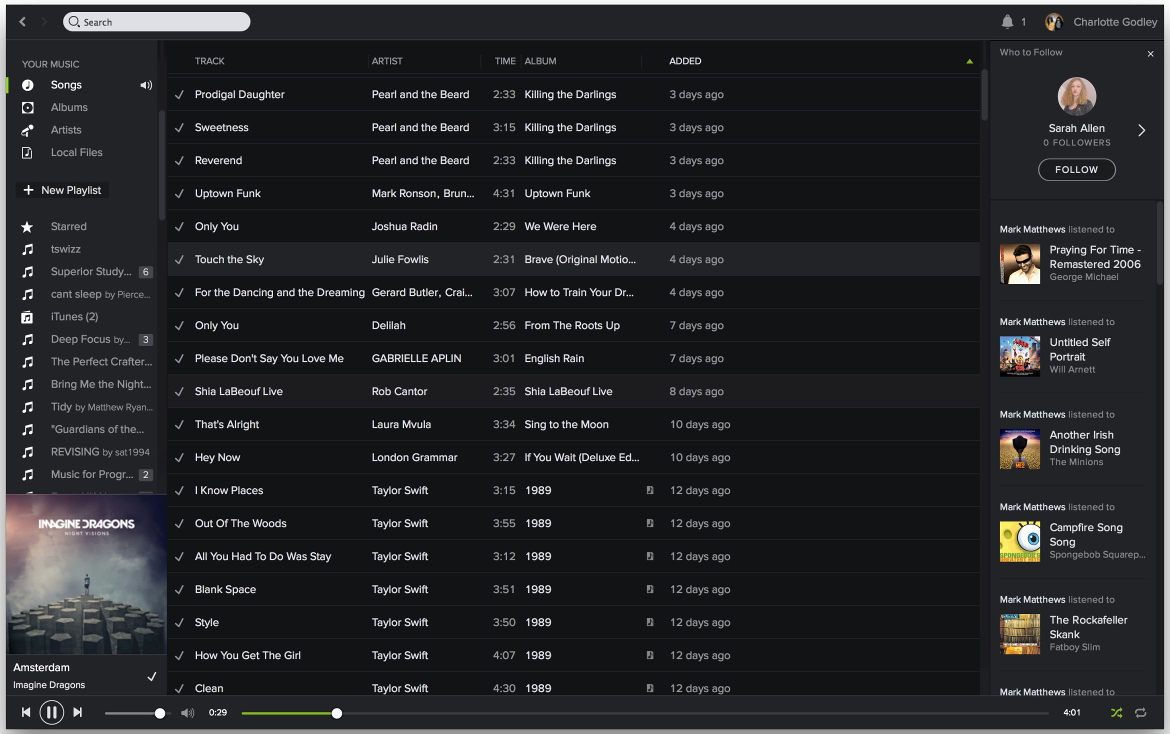
4.1.1 Class diagrams and mind map

The class structure design process started with a mind map to initially appraise the connection between each musical symbol. This helped break down a piece from a musician's view, and showed what information would be necessary between each symbol's class.

An initial class diagram was drawn from the mind map. This was modified during the course of development, testing and research of the initial model. In particular, other sources such as Music21 were looked at, a toolkit for computer-aided musicology (MIT, 2013), which helped to examine whether the initial model was missing any classes or attributes. Furthermore, development showed that particular methods, such as the toString method and the Lilypond method could be improved by using inheritance from base classes in order to avoid unnecessary code duplication. The diagrams described are in the appendices.

4.2 UI Design

The User interface for this project is designed after looking at the user interfaces used by other music applications. In particular, the developer was inspired by Spotify, shown in figure 10.



TRACK	ARTIST	TIME	ALBUM	ADDED
Prodigal Daughter	Pearl and the Beard	2:33	Killing the Darlings	3 days ago
Sweetness	Pearl and the Beard	3:15	Killing the Darlings	3 days ago
Reverend	Pearl and the Beard	2:33	Killing the Darlings	3 days ago
Uptown Funk	Mark Ronson, Brun...	4:31	Uptown Funk	3 days ago
Only You	Joshua Radin	2:29	We Were Here	4 days ago
Touch the Sky	Julie Fowlis	2:31	Brave (Original Motio...	4 days ago
For the Dancing and the Dreaming	Gerard Butler, Crai...	3:07	How to Train Your Dr...	4 days ago
Only You	Delilah	2:56	From The Roots Up	7 days ago
Please Don't Say You Love Me	GABRIELLE APLIN	3:01	English Rain	7 days ago
Shia LaBeouf Live	Rob Cantor	2:35	Shia LaBeouf Live	8 days ago
That's Alright	Laura Mvula	3:34	Sing to the Moon	10 days ago
Hey Now	London Grammar	3:27	If You Wait (Deluxe Ed...	10 days ago
I Know Places	Taylor Swift	3:15	1989	12 days ago
Out Of The Woods	Taylor Swift	3:55	1989	12 days ago
All You Had To Do Was Stay	Taylor Swift	3:12	1989	12 days ago
Blank Space	Taylor Swift	3:51	1989	12 days ago
Style	Taylor Swift	3:50	1989	12 days ago
How You Get The Girl	Taylor Swift	4:07	1989	12 days ago
Clean	Taylor Swift	4:30	1989	12 days ago

Figure 10: Spotify user interface

4.2.1 Main Display

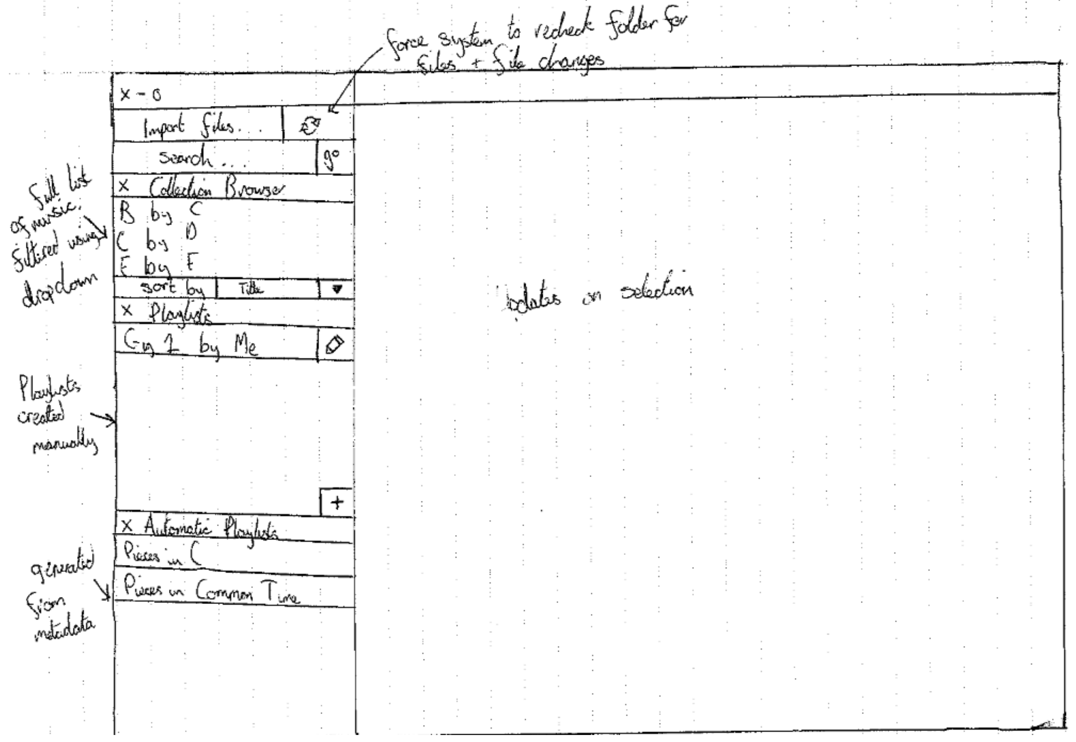


Figure 11: Main User interface of the project

Figure 11 shows the main view of the application. Various panes to the left can be closed using the X button and show different ways the music can be displayed, either as individual units or as playlists. The larger pane next to it shows the area in which sheet music or a list of pieces in a playlist will be displayed, depending on the selection from the left pane. Updates to this and pop up boxes displaying dependent on buttons in the window are displayed and explained in the appendices.

4.2.2 Musician feedback survey

In order to understand how well this user interface works with a variety of users, a survey was designed which will be given to a selection of musicians, who will feedback on how easy the UI is to use and any updates which should be made to improve it. This feedback session will be performed after the initial sketches are made into a virtual user interface with no back end connected to the buttons. An example survey is provided in the appendices.

4.3 Test Design

This project will be developed using Test Driven Development. This is an Agile software development methodology which utilises the rules that a line of code should not be written unless there is a failing automated test (Newkirk and Vorontsov, 2004). This methodology has been chosen as the nature of the notation of music means that meticulous detail must be payed to how and with what symbol every element is notated, and Test Driven Development will significantly improve the quality of the software by closely integrating testing with the development process.

Development of tests and production code are ongoing, and as such the tests confirm critical, but self-contained units are correct, such as an accent being added to a measure correctly, or a note's pitch being created with a particular note name or octave number.

Test cases were created using the aforementioned software MuseScore. These were produced by creating a file for every area of notation (e.g. clefs, time signatures, note durations, pitch) and applying each and every symbol possible within that area to the music. It was decided to create test cases in this way to thoroughly ensure that no piece of notation was missed. An example testcase, and a list of all testcases in use and their value in real world testing, are included in the appendices.

5 Project Management Review

5.1 Current progress

During the initial stages of the project, time was dedicated to researching the appropriate language to use, the file format and the methods and algorithms used by other packages. This involved looking at the projects done in the field of music in Python previously, such as Music21 (MIT, 2013), and other projects listed on the Python Foundation website (The Python Foundation, 2015).

Many important decisions were made from this research period, such as the decision to use Lilypond to typeset music files rather than create a new algorithm and the research of MusicOCR options available, leading to the decision that MusicOCR is too big a topic for this project to create a new algorithm.

After this research period, class diagrams were drawn and some initial code implementation for the rendering and metadata objectives was developed. It was decided after this initial stage to use Test Driven Development, as the code base and algorithm for loading in a music file was becoming hard to confirm crucial details were being parsed correctly. A set of unit tests were written for the initial implementation, and from this point onward the methodology was applied.

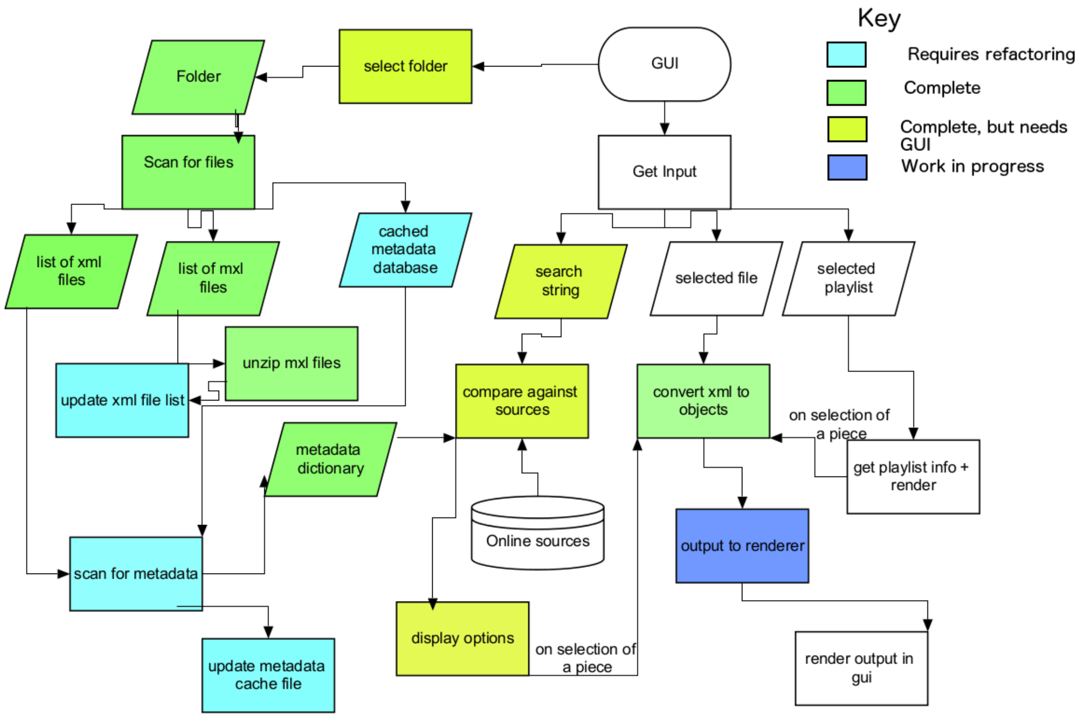


Figure 12: Flowchart colour coded according to progress

Figure 12 shows the flowchart shown in figure 9 colour coded to show progress.

Of the areas shaded in light blue, to indicate the area needs refactoring, the cached metadata database is currently stored as a serialised python object. In order to make this as extendible and portable as possible, this will need to be refactored to using an SQLite file to be considered completed, as stipulated in section 3.3.3. SQLite is a light implementation of an SQL database stored as a single file, which should be relatively simple to implement in other languages as it is standardised.

5.2 Adjustments made

During the initial planning phase the developer included coursework deadlines. However, concessions were not considered at times when other coursework should and did take precedence. In particular, the week before the Languages and Compilers coursework deadline, work was solely focussed on development for this coursework.

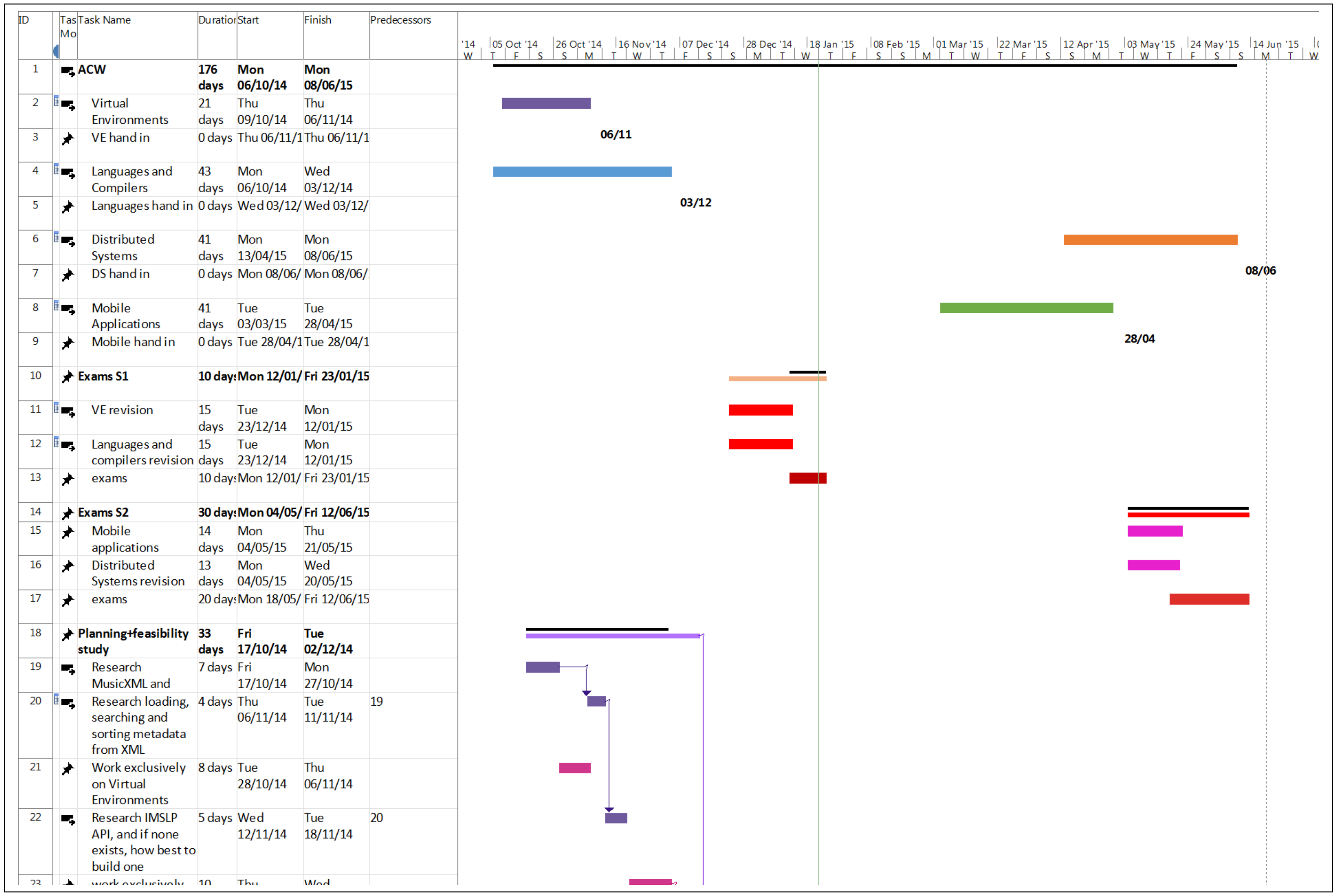
Furthermore, the initial research phase was very open ended and some topics took less time than was intended than others. In this case, the developer used the extra time working on other coursework, though in the future adjustments to move future tasks forward should be done instead.

Thirdly, the development methodology in use now is Test Driven Development, meaning that the testing phases between prototype developments can be changed to specified testing sections, such as user interface testing and stress testing, rather than functionality testing.

Finally, the first two weeks of term were affected by jet lag from attending a conference in the United States, which meant that documentation and development suffered a little. Again, the developer was aware that this could occur and should have considered it as a potential risk, though this particular problem will not occur again in the course of the project.

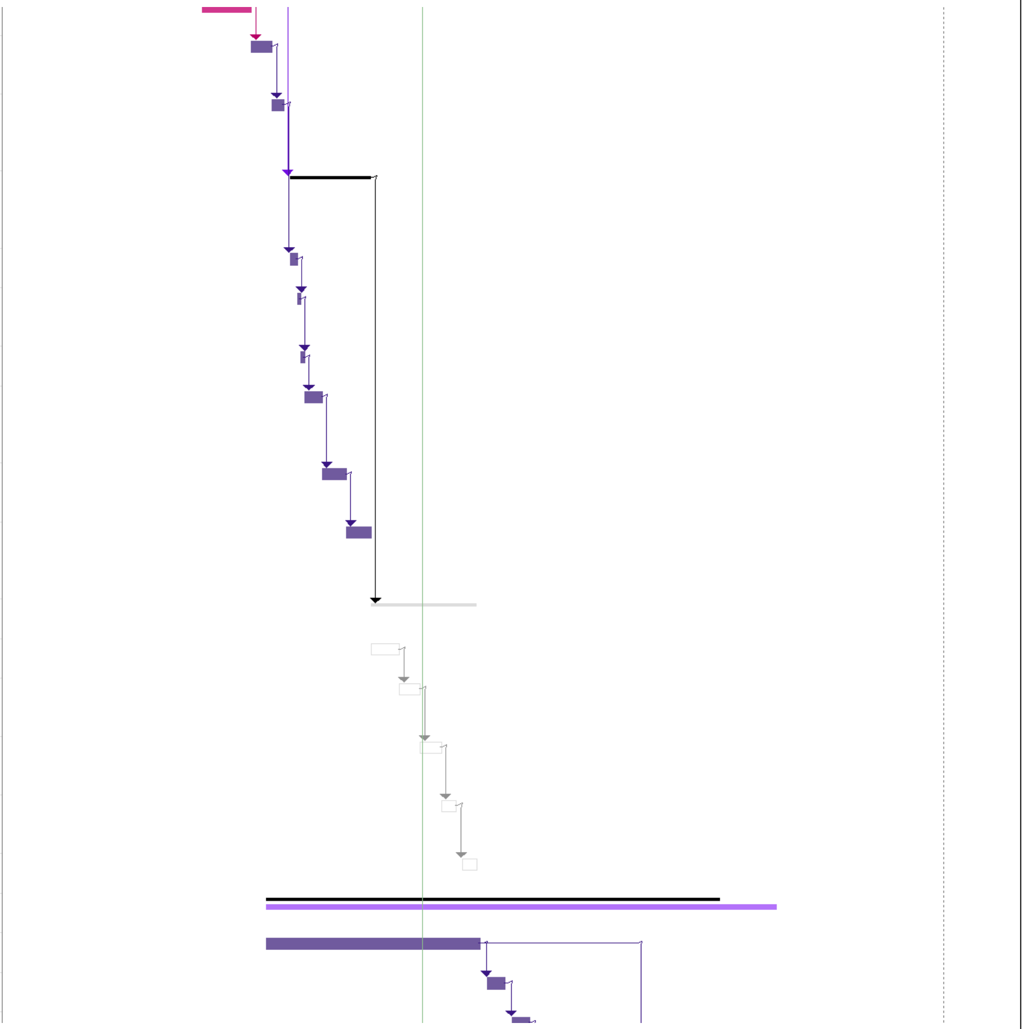
Considering these factors, the revised time plan in figure 13 includes time dedicated solely to coursework for other modules where deadlines arise, and attempts to more clearly define tasks which are open ended.

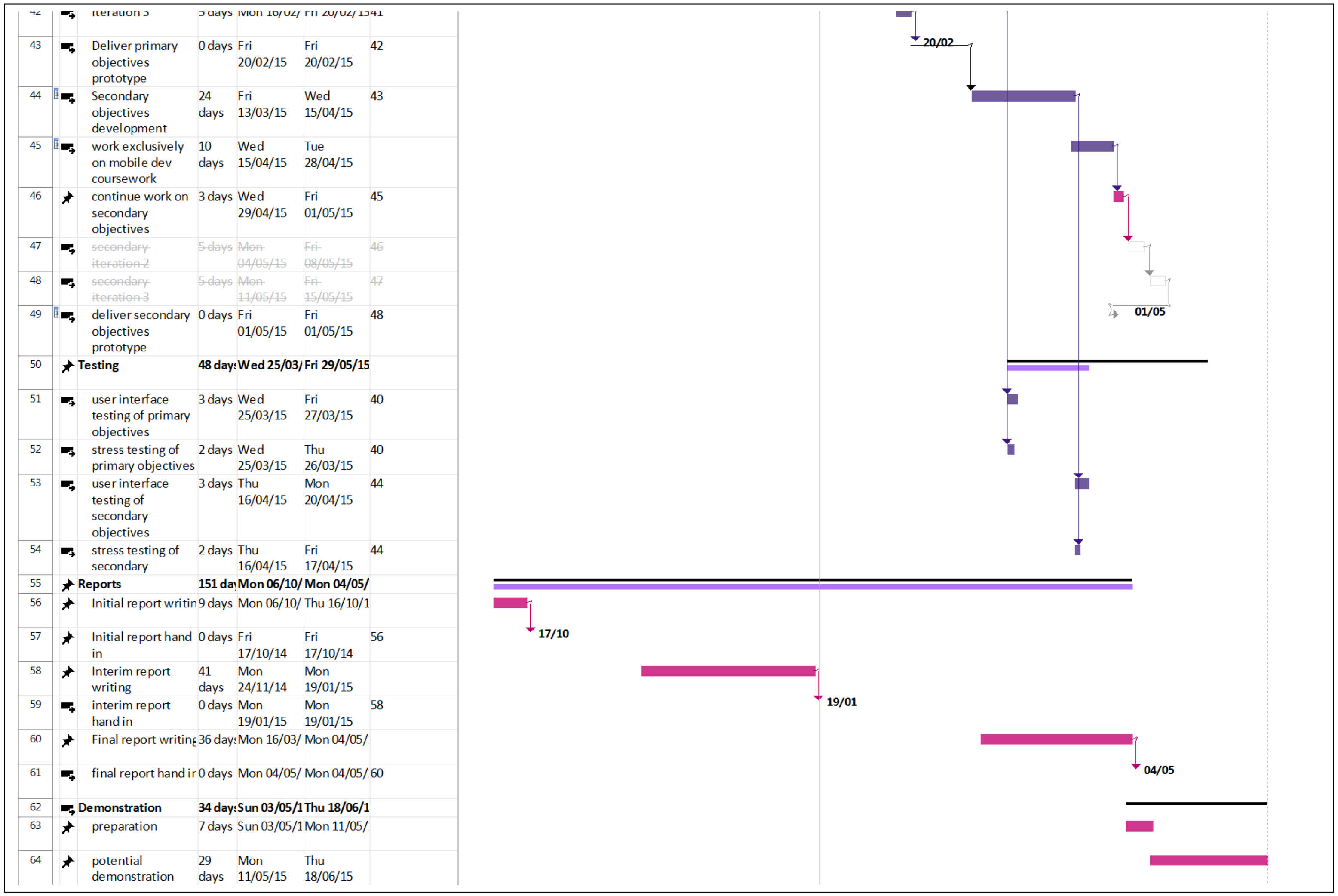
5.3 Revised timeplan



ID	Tas Mo	Task Name	Duratior	Start	Finish	Predecessors
1		**ACW**	**176 days**	**Mon 06/10/14**	**Mon 08/06/15**	
2		Virtual Environments	21 days	Thu 09/10/14	Thu 06/11/14	
3		VE hand in	0 days	Thu 06/11/1	Thu 06/11/1	
4		Languages and Compilers	43 days	Mon 06/10/14	Wed 03/12/14	
5		Languages hand in	0 days	Wed 03/12/	Wed 03/12/	
6		Distributed Systems	41 days	Mon 13/04/15	Mon 08/06/15	
7		DS hand in	0 days	Mon 08/06/	Mon 08/06/	
8		Mobile Applications	41 days	Tue 03/03/15	Tue 28/04/15	
9		Mobile hand in	0 days	Tue 28/04/1	Tue 28/04/1	
10		**Exams S1**	**10 day:**	**Mon 12/01/**	**Fri 23/01/15**	
11		VE revision	15 days	Tue 23/12/14	Mon 12/01/15	
12		Languages and compilers revision	15 days	Tue 23/12/14	Mon 12/01/15	
13		exams	10 day:	Mon 12/01/	Fri 23/01/15	
14		**Exams S2**	**30 day:**	**Mon 04/05/**	**Fri 12/06/15**	
15		Mobile applications	14 days	Mon 04/05/15	Thu 21/05/15	
16		Distributed Systems revision	13 days	Mon 04/05/15	Wed 20/05/15	
17		exams	20 day:	Mon 18/05/	Fri 12/06/15	
18		**Planning+feasibility study**	**33 days**	**Fri 17/10/14**	**Tue 02/12/14**	
19		Research MusicXML and	7 days	Fri 17/10/14	Mon 27/10/14	
20		Research loading, searching and sorting metadata from XML	4 days	Thu 06/11/14	Tue 11/11/14	19
21		Work exclusively on Virtual Environments	8 days	Tue 28/10/14	Thu 06/11/14	
22		Research IMSLP API, and if none exists, how best to build one	5 days	Wed 12/11/14	Tue 18/11/14	20
23		work exclusively	10	Thu	Wed	

23	work exclusively on compilers	10 days	Thu 20/11/14	Wed 03/12/14	
24	Research MIDI and sound generation in chosen	4 days	Thu 04/12/14	Tue 09/12/14	23
25	Research Music OCR techniques and validate whether it can be	3 days	Wed 10/12/14	Fri 12/12/14	24
26	**Design and feedback**	**17 days**	**Mon 15/12/14**	**Tue 06/01/15**	**18**
27	Design suitable user interface	2 days	Mon 15/12/14	Tue 16/12/14	25
28	Design survey form to show other musicians	1 day	Wed 17/12/14	Wed 17/12/14	27
29	give UI and survey to other musicians	1 day	Thu 18/12/14	Thu 18/12/14	28
30	Design data structures and file structure for metadata	3 days	Fri 19/12/14	Tue 23/12/14	29
31	design searching algorithm for metadata	5 days	Wed 24/12/14	Tue 30/12/14	30
32	design IMSLP metadata extraction algorithm	5 days	Wed 31/12/14	Tue 06/01/15	31
33	~~Test implementation~~	~~22 days~~	~~Wed 07/01/15~~	~~Thu 05/02/15~~	~~26~~
34	~~design+implement tests for rendering~~	~~6 days~~	~~Wed 07/01/15~~	~~Wed 14/01/15~~	
35	~~design+implement tests for metadata generation~~	~~4 days~~	~~Thu 15/01/15~~	~~Tue 20/01/15~~	~~34~~
36	~~Design+implement tests for searching algorithm~~	~~4 days~~	~~Wed 21/01/15~~	~~Mon 26/01/15~~	~~35~~
37	~~Design+ implement tests for IMSLP~~	~~4 days~~	~~Tue 27/01/15~~	~~Fri 30/01/15~~	~~36~~
38	~~Design+implement tests for UI~~	~~4 days~~	~~Mon 02/02/15~~	~~Thu 05/02/15~~	~~37~~
39	**Software implementation**	**93 days**	**Mon 08/12/14**	**Wed 15/04/15**	
40	Primary objectives development	45 days	Mon 08/12/14	Fri 06/02/15	
41	primary iteration 2	5 days	Mon 09/02/	Fri 13/02/15	40
42	iteration 3	5 days	Mon 16/02/	Fri 20/02/15	41





42	iteration 3	5 days	Mon 16/02/	Fri 20/02/15	41
43	Deliver primary objectives prototype	0 days	Fri 20/02/15	Fri 20/02/15	42
44	Secondary objectives development	24 days	Fri 13/03/15	Wed 15/04/15	43
45	work exclusively on mobile dev coursework	10 days	Wed 15/04/15	Tue 28/04/15	
46	continue work on secondary objectives	3 days	Wed 29/04/15	Fri 01/05/15	45
47	~~secondary iteration 2~~	~~5 days~~	~~Mon 04/05/15~~	~~Fri 08/05/15~~	~~46~~
48	~~secondary iteration 3~~	~~5 days~~	~~Mon 11/05/15~~	~~Fri 15/05/15~~	~~47~~
49	deliver secondary objectives prototype	0 days	Fri 01/05/15	Fri 01/05/15	48
50	**Testing**	**48 days**	**Wed 25/03/**	**Fri 29/05/15**	
51	user interface testing of primary objectives	3 days	Wed 25/03/15	Fri 27/03/15	40
52	stress testing of primary objectives	2 days	Wed 25/03/15	Thu 26/03/15	40
53	user interface testing of secondary objectives	3 days	Thu 16/04/15	Mon 20/04/15	44
54	stress testing of secondary	2 days	Thu 16/04/15	Fri 17/04/15	44
55	**Reports**	**151 day**	**Mon 06/10/**	**Mon 04/05/**	
56	Initial report writin	9 days	Mon 06/10/	Thu 16/10/1	
57	Initial report hand in	0 days	Fri 17/10/14	Fri 17/10/14	56
58	Interim report writing	41 days	Mon 24/11/14	Mon 19/01/15	
59	interim report hand in	0 days	Mon 19/01/15	Mon 19/01/15	58
60	Final report writing	36 days	Mon 16/03/	Mon 04/05/	
61	final report hand ir	0 days	Mon 04/05/	Mon 04/05/	60
62	**Demonstration**	**34 days**	**Sun 03/05/1**	**Thu 18/06/1**	
63	preparation	7 days	Sun 03/05/1	Mon 11/05/	
64	potential demonstration	29 days	Mon 11/05/15	Thu 18/06/15	

Figure 13: Updated timeplan

Appendices

A Mind map of elements of Music

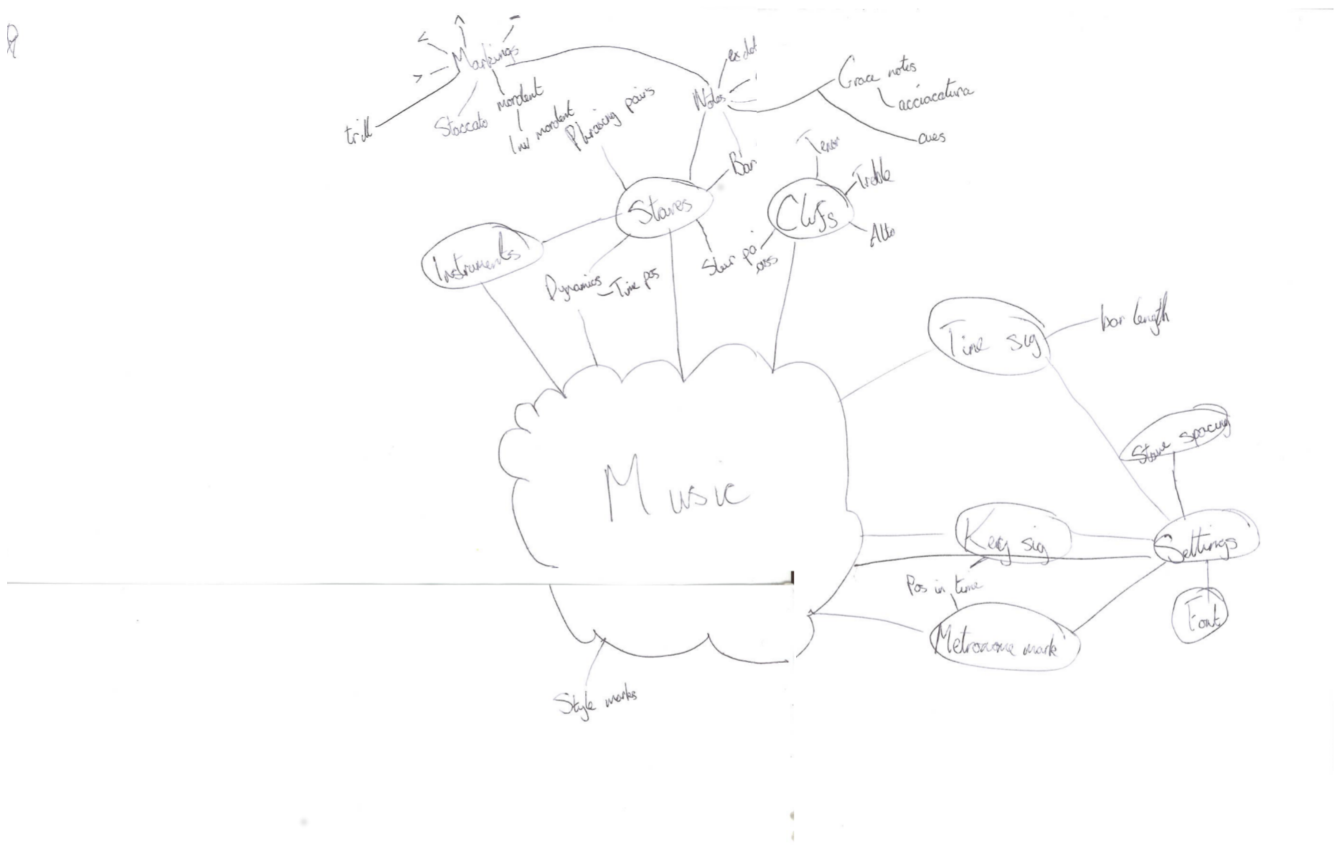


Figure 14: A hand drawn mind map of elements of music

B Initial class diagram

Figure 15 shows the class diagram derived from the mind map in figure 14. This was modified and changed when the classes were implemented in code as various things such as the rendering output methods and the toString methods involved code duplication without the use of a base class for the majority of the objects.

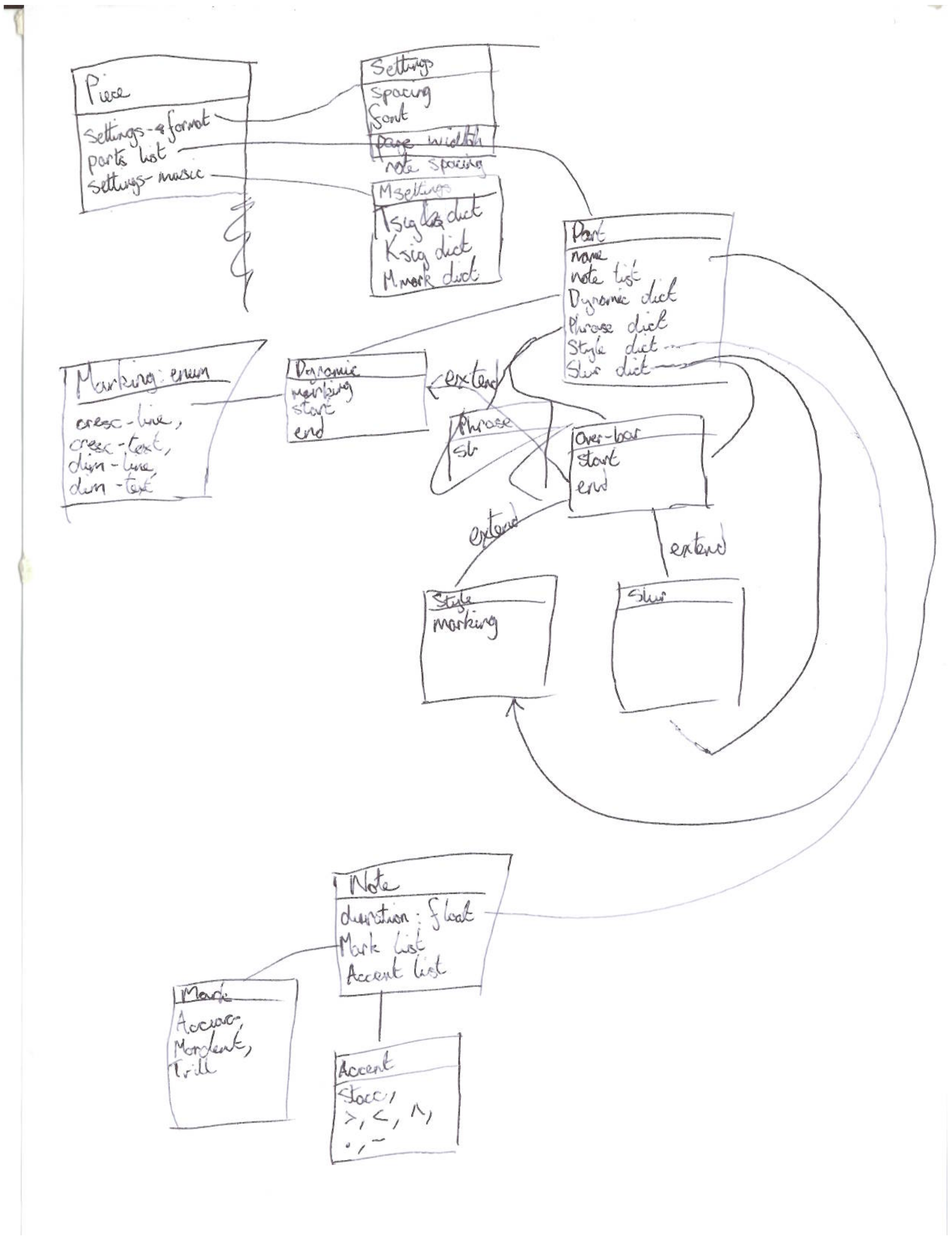


Figure 15: A class diagram based on the initial mind map

C Revised computerised class diagram

C.1 Classes inheriting from Base

Figures C.2, 17, 18, and 19 show all of the classes which inherit from Base. They inherit from base because the debugging "toString" method is common to all of them, and ensures that problems in using duck typing do not occur when calling the toLily method.

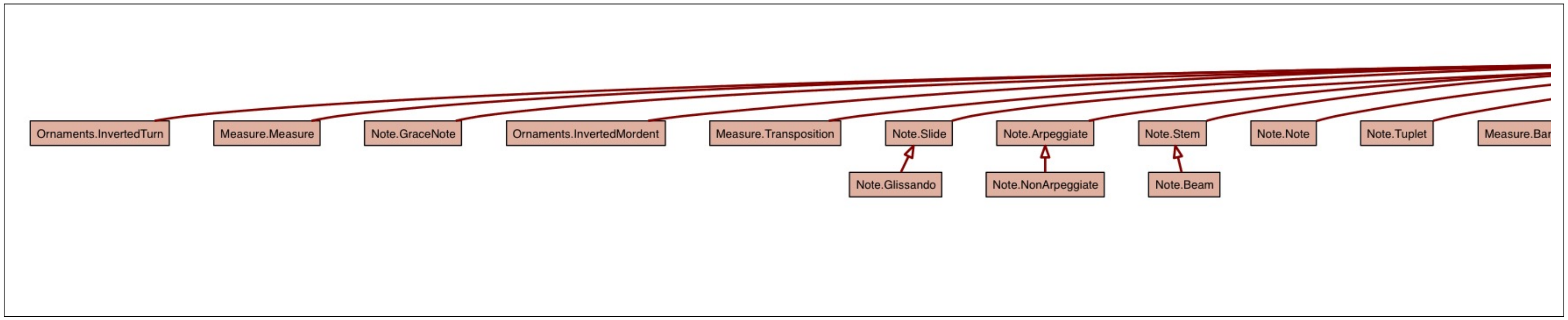


Figure 16: Classes inheriting from Base

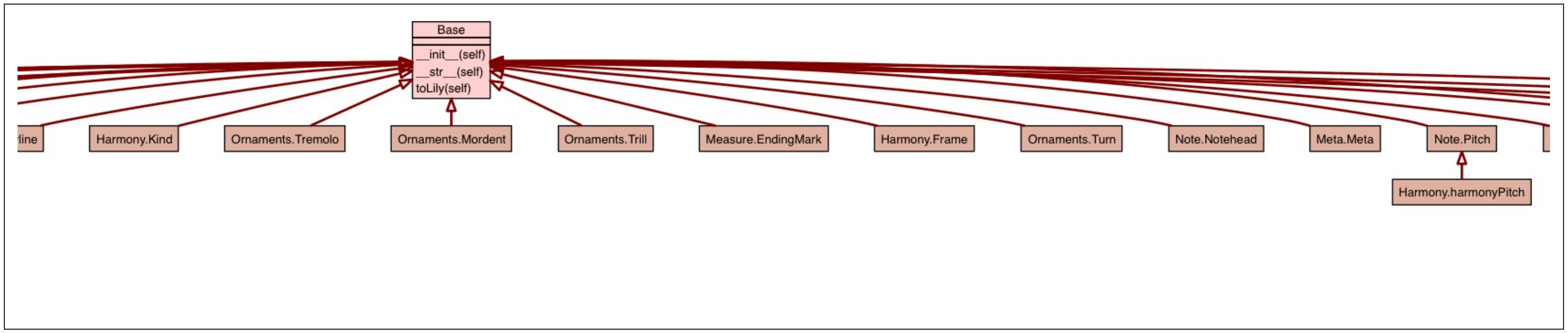


Figure 17: Classes inheriting from base

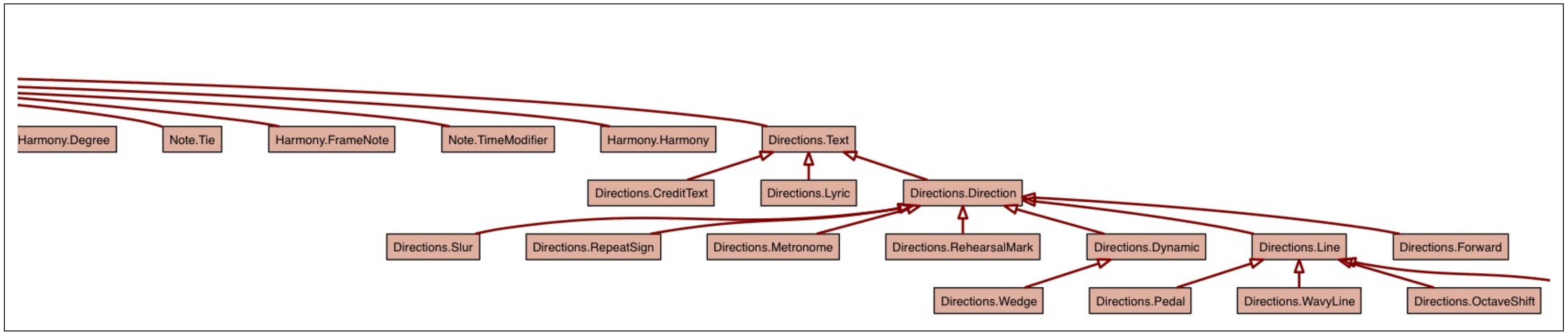


Figure 18: Classes inheriting from Base

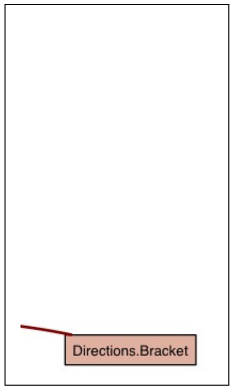


Figure 19: Classes inheriting from Base

C.2 Notation classes

The classes in figure 20 inherit from the Notation class, for the same reasoning as those in the diagram in figure inherit from the base class. That is to say, each class has a common toString method.

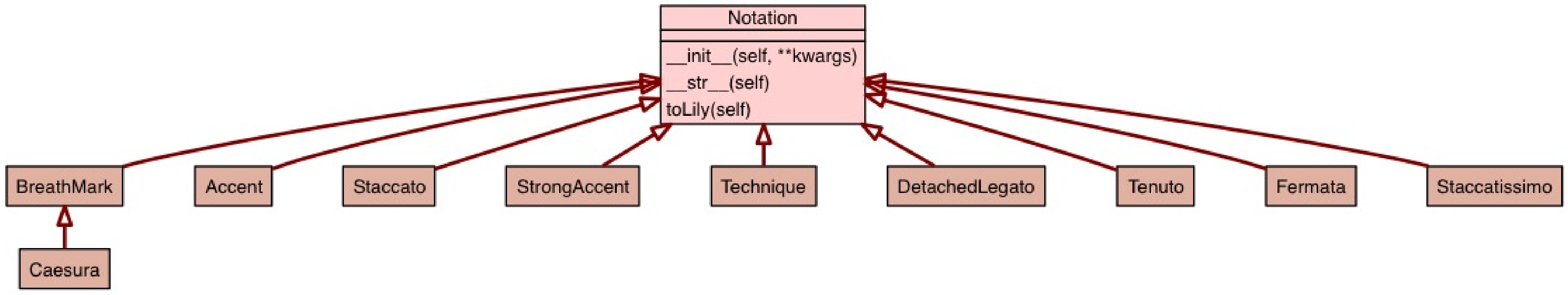


Figure 20: Classes inheriting from Notation

C.3 Exception classes

Figures 21, 22, 23 are classes created to indicate to the developer that an exception has been thrown whilst parsing a musicXML file. The first indicates that no measure ID has been found, which means directions and notations cannot be added as the program does not know which measure to add them to. The second indicates that a part ID has been found which has not yet been created in the piece class when it should have, and the third indicates that no score-part ID has been found, which causes similar problems as the first exception indicates.

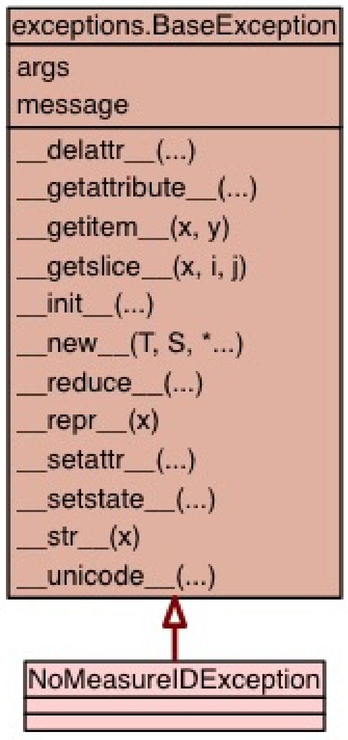


Figure 21: MeasureID exception class

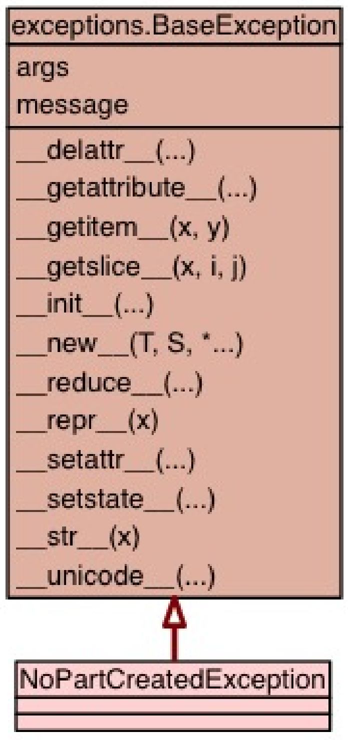


Figure 22: Part exception class

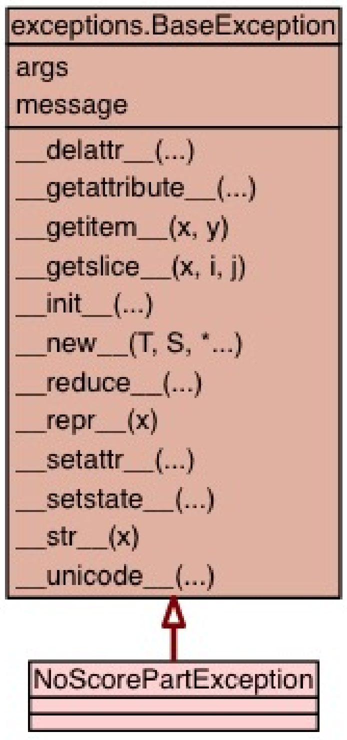


Figure 23: ScorePart exception class

C.4 Standalone classes

Figures 24, 25, 26, 27, 28, and 29 do not use inheritance, as their string methods and toLily methods have no connection or similarity to any other class in the system.

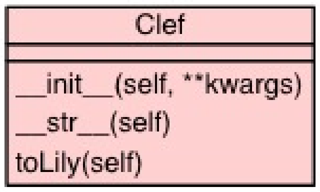


Figure 24: Clef class

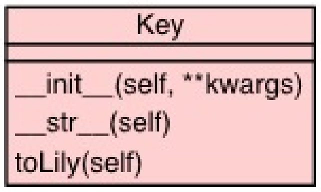


Figure 25: Key class

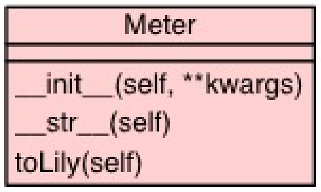


Figure 26: Meter class

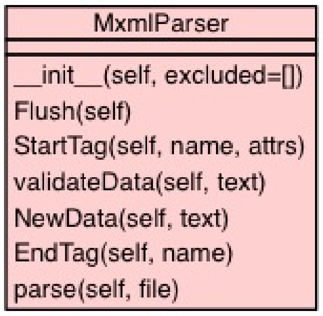


Figure 27: The main MxmlParser class, which applies handlers to each tag

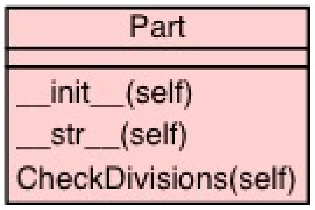


Figure 28: Part class, which contains measures

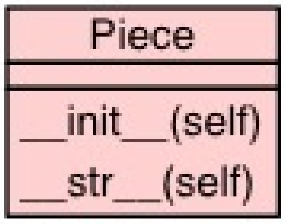


Figure 29: Piece class, which contains parts

D Initial User Interface Design

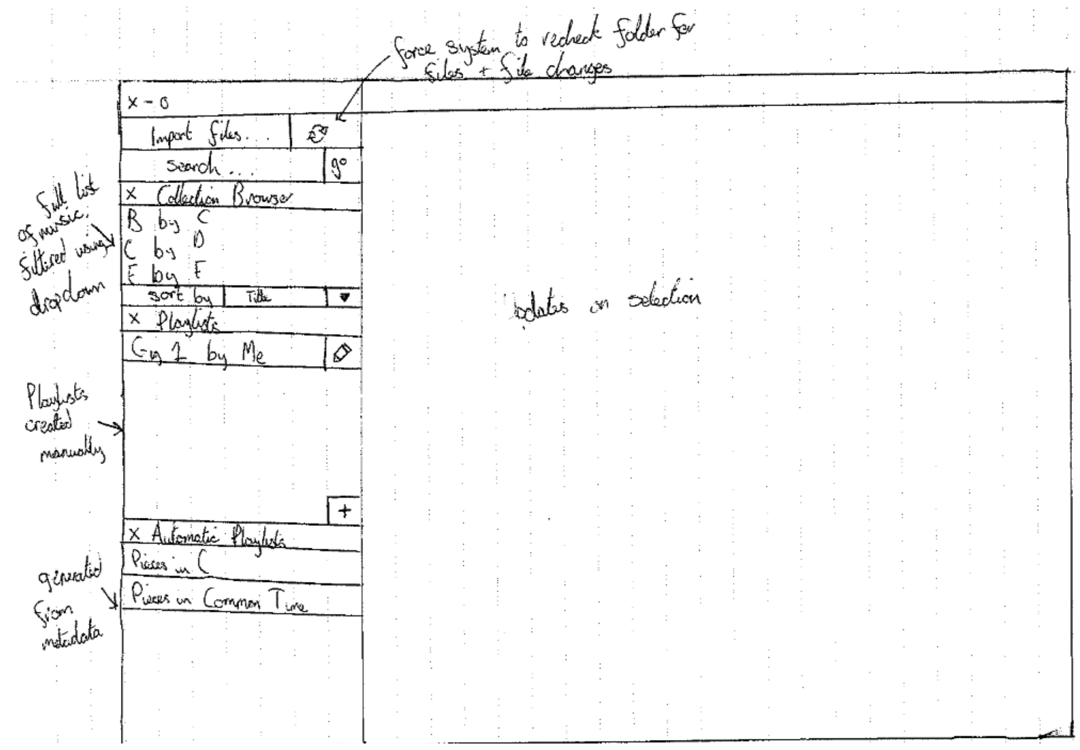


Figure 30: The main GUI

Figure 30 shows the main graphical user interface. Figure 31 shows the popup window displayed when "import files" is clicked. Figure 32 shows the popup window displayed when the plus button on the playlists pane is clicked.

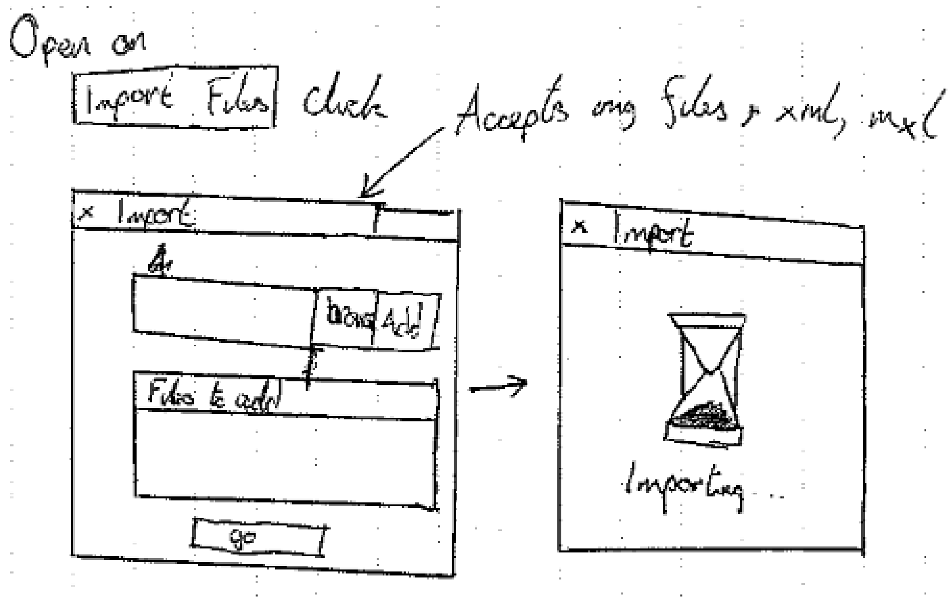


Figure 31: Popup box that opens when the user clicks "import files"

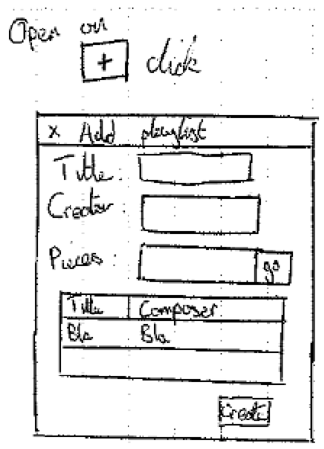


Figure 32: Popup box on click of the plus button, creating a new playlist

D.1 Changes to main display when sheet music selected

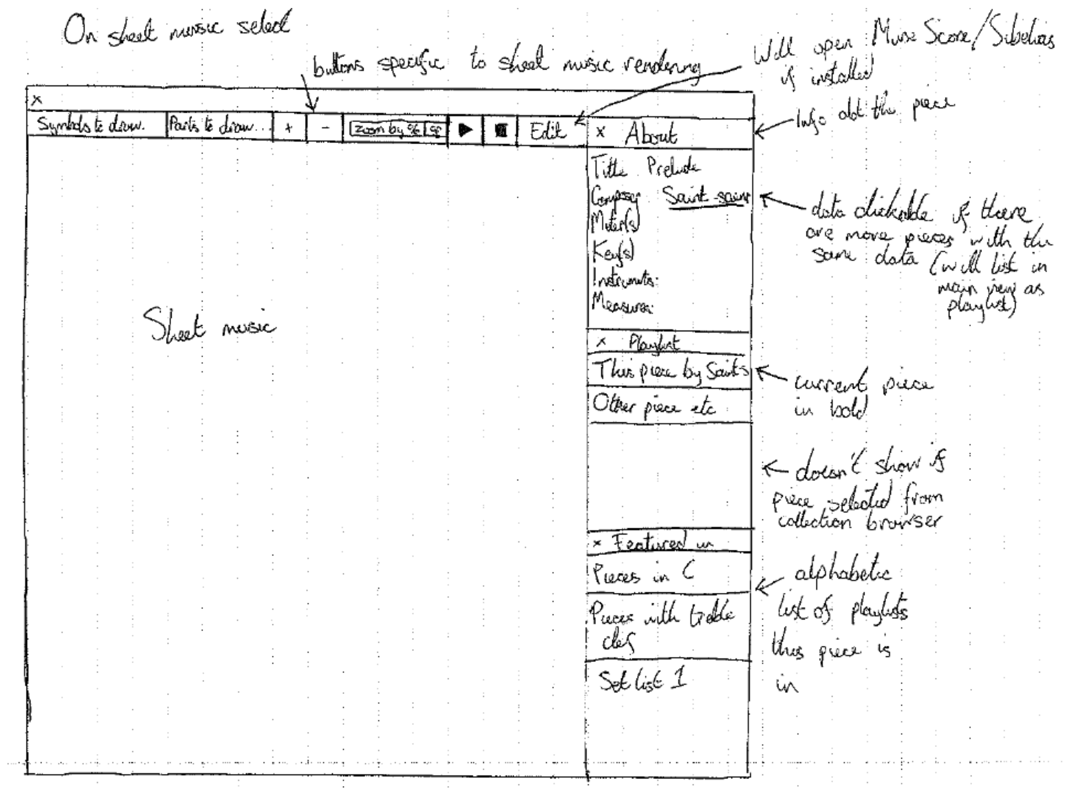


Figure 33: The main pane when sheet music is selected

The image shown in figure 33 shows the updated main pane of figure 30 when a piece of music has been selected. The smaller windows to the right show an about pane, which displays all information about the piece itself, a "featured in" pane which will list all playlists containing this piece, and a "playlist" pane.

Within the about pane, if a piece of data such as "Saint-Saens" in figure 33 is underlined, it can be clicked which will lead to a playlist of other pieces which contain the same data.

The playlist pane will only display if the piece has been selected for viewing from an existing playlist. That is to say, if this piece were to be clicked from the collection browser shown in figure 30, this window would not be open.

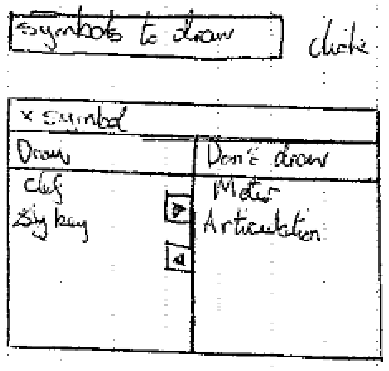


Figure 34: The pop up pane displayed when symbols to draw is clicked

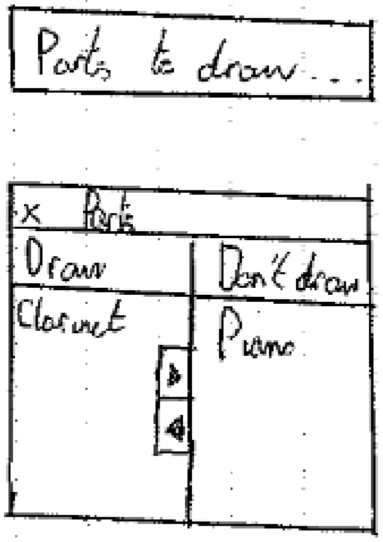


Figure 35: the popup pain displayed when parts to draw is clicked

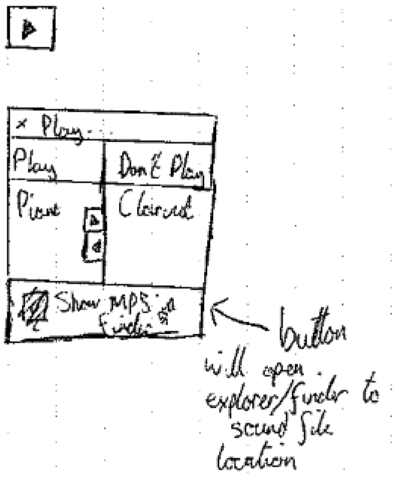


Figure 36: the popup pain displayed when the triangular play button is clicked

The figures 34, 35, and 36 show the extra windows which display when "symbols to draw", "parts to draw" and the triangle (play) button are clicked, which are buttons displayed above the main pane in figure 33. Whilst not an objective or an intended benefit, the symbols window is included in order to help users who may not be fully confident reading sheet music which includes symbols they do not understand.

D.2 Changes to main display when playlist is selected

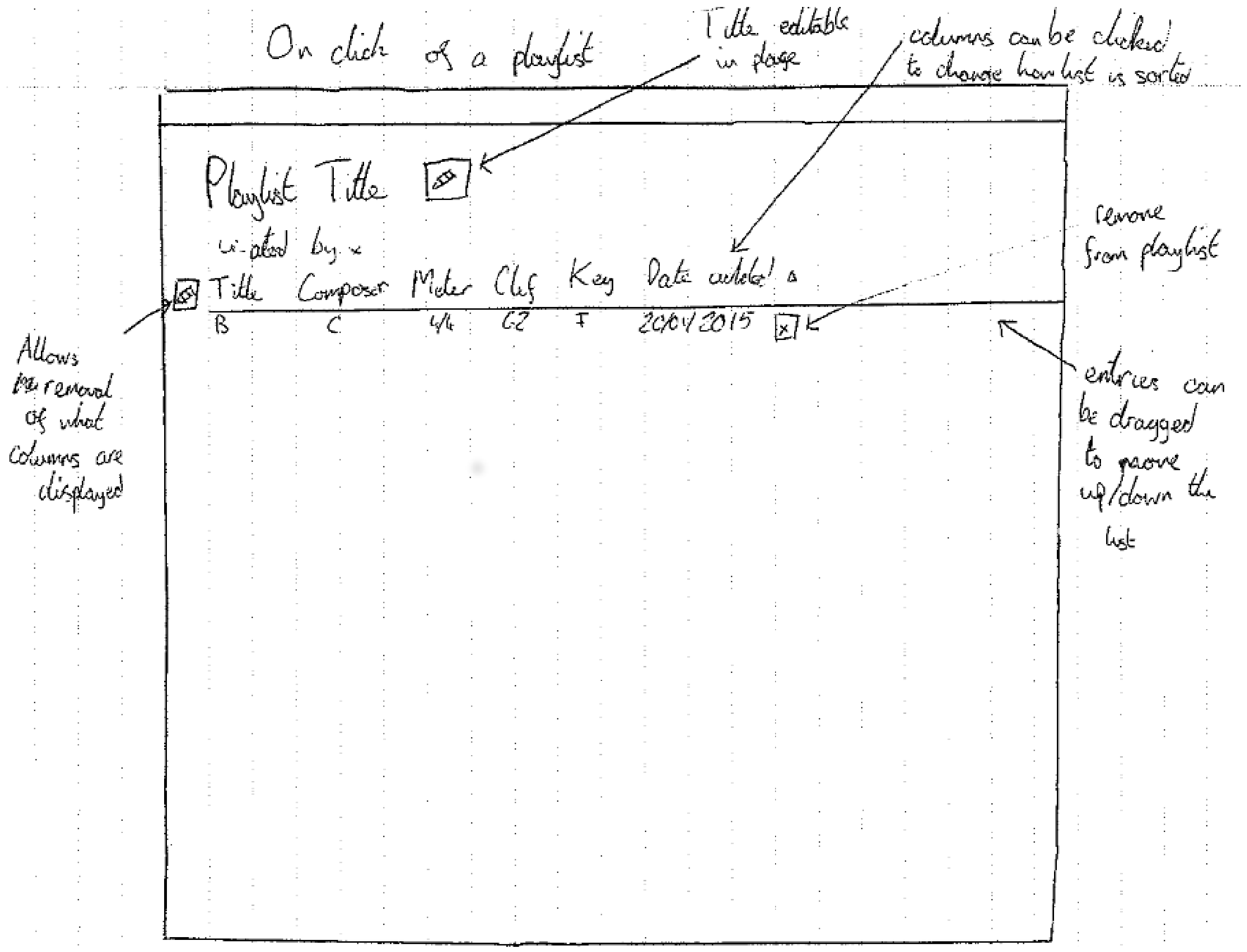

On click of a playlist
Title editable in place
columns can be clicked to change how list is sorted
Playlist Title
created by x
Title Composer Meter Clef Key Date added
B C 4/4 G2 F 20/04/2015
remove from playlist
Allows the removal of what columns are displayed
entries can be dragged to move up/down the list

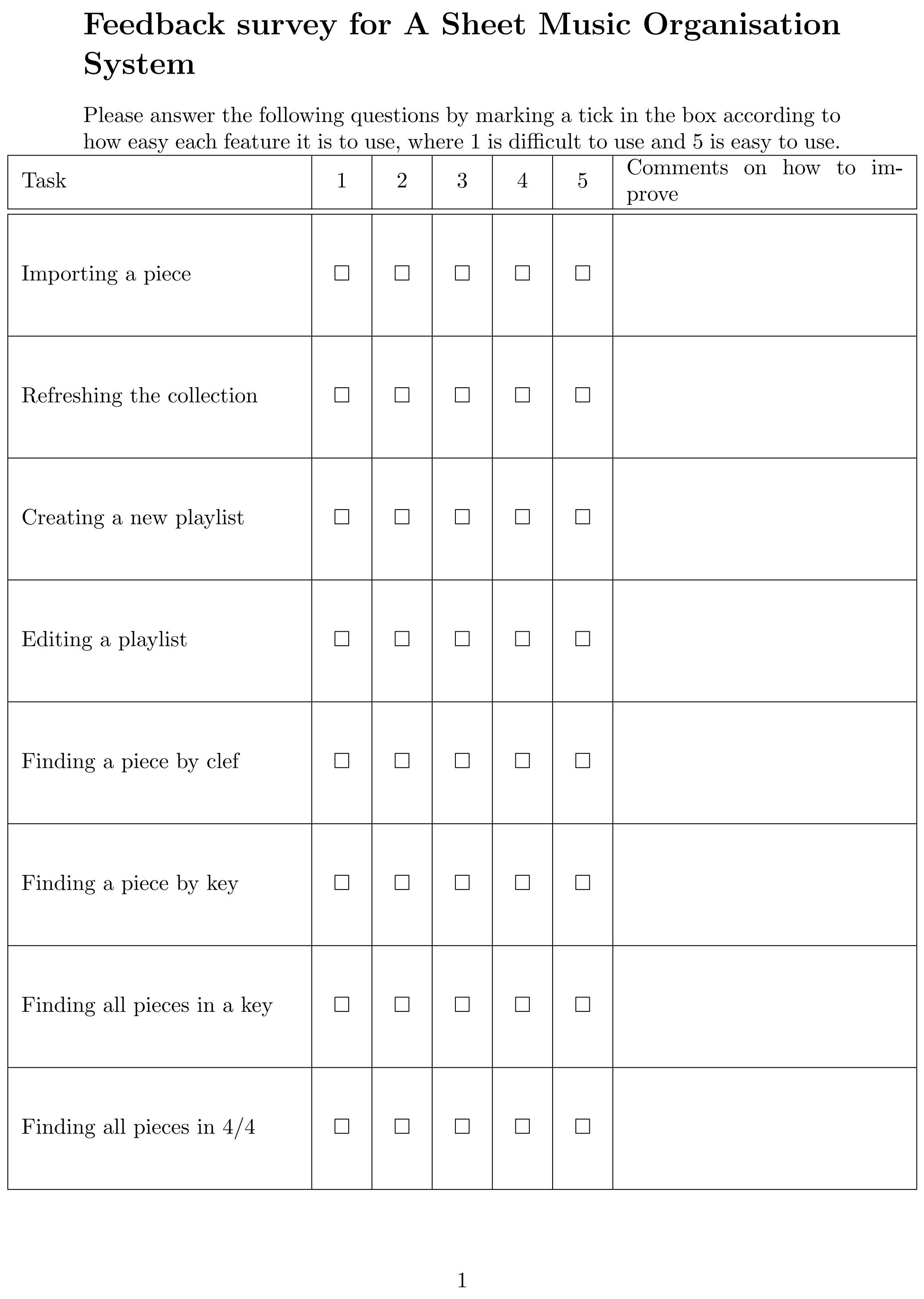
Figure 37: The main pane of the GUI when a playlist is selected

Figure 37 shows the main pane of figure 30 when a playlist has been selected. This can occur by clicking a manually created playlist from the second pane in figure 30, an auto-generated playlist from the third pane in the same figure, or by clicking on underlined text in the about pane of figure 33.

The buttons which appear like pens allow the user to edit the title of the playlist or change which columns are displayed - both of these are done in place, that is to say, no extra pop up boxes are needed for the user to change anything in this window. Further to this, the user may move pieces up or down the playlist by dragging the item, and delete them using the "x" button.

E User Interface Feedback survey

Figure 38 shows an example blank survey given to participants to analyse the usefulness of the current user interface. This is basic, as functionality like searching at the initial survey stage will not have been connected to the user interface itself.



Feedback survey for A Sheet Music Organisation System

Please answer the following questions by marking a tick in the box according to how easy each feature it is to use, where 1 is difficult to use and 5 is easy to use.

Task	1	2	3	4	5	Comments on how to improve
Importing a piece	☐	☐	☐	☐	☐	
Refreshing the collection	☐	☐	☐	☐	☐	
Creating a new playlist	☐	☐	☐	☐	☐	
Editing a playlist	☐	☐	☐	☐	☐	
Finding a piece by clef	☐	☐	☐	☐	☐	
Finding a piece by key	☐	☐	☐	☐	☐	
Finding all pieces in a key	☐	☐	☐	☐	☐	
Finding all pieces in 4/4	☐	☐	☐	☐	☐	

1

Figure 38: The survey given to users to analyse how easy it is to use the interface

F Test cases

Table 4 gives a brief explanation of all the current testcases being used to validate the system against real world applications. Listing 1 gives one example, keySignatures.xml, of a musicXML file.

Name	Purpose
Accidentals.xml	Tests system properly handles all possible accidentals attached to notes
GraceNotes.xml	Tests gracenotes attached to notes
Tremolo.xml	tests tremolo on notes
TrillsFermataOrnaments.xml	tests trills, fermatas (pauses) and other ornaments on notes
arpeggiosAndGlissandos.xml	tests arpeggios and glissandos on notes
barlines.xml	tests different barlines applied to measures
beams.xml	tests beaming of notes (quavers, semi quavers etc)
breathMarks.xml	tests breathmark notation next to notes
clefs.xml	tests all possible clef types
duration_and_stem_direction.xml	tests duration of notes and their stem (stick) direction)
dynamics.xml	tests dynamics (loud and quiet) and their position in a measure
fingering.xml	fingering notation specific to string instruments
keySignatures.xml	tests all key signatures and their names are correct
lines.xml	tests a variety of lines over bars, such as pedal marks and repeat alternative bars
multiple_parts.xml	tests how the system handles more than one instrumental part
noteheads.xml	tests all possible changes to the shape of the notehead
repeatMarks.xml	tests repeat marks are loaded correctly (i.e, points at which the player must go back to a specific sign and play it again)
text.xml	tests text markings, such as dynamic and tempo markings like "andante", but also things like lyrics
tuplets.xml	tests tuplets, where a note must be played differently to how it is notated according to it's tuplet value
two_staves_one_part.xml	tests loading of two staves in one part, such as pianos and harpsichords

Table 4: All testcases currently in use

Listing 1: key signature testcase

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE score-partwise PUBLIC "-//Recordare//DTD MusicXML 2.0 Partwise//EN" "http://www.
    musicxml.org/dtds/partwise.dtd">
<score-partwise>
  <identification>
    <encoding>
      <software>MuseScore 1.3</software>
      <encoding-date>2014-12-22</encoding-date>
      </encoding>
    </identification>
  <defaults>
    <scaling>
      <millimeters>7.05556</millimeters>
      <tenths>40</tenths>
      </scaling>
    <page-layout>
      <page-height>1683.78</page-height>
      <page-width>1190.55</page-width>
      <page-margins type="even">
        <left-margin>56.6929</left-margin>
        <right-margin>56.6929</right-margin>
        <top-margin>56.6929</top-margin>
        <bottom-margin>113.386</bottom-margin>
        </page-margins>
      <page-margins type="odd">
        <left-margin>56.6929</left-margin>
        <right-margin>56.6929</right-margin>
        <top-margin>56.6929</top-margin>
        <bottom-margin>113.386</bottom-margin>
        </page-margins>
      </page-layout>
    </defaults>
  <part-list>
    <score-part id="P1">
      <part-name>Flute</part-name>
      <part-abbreviation>Fl.</part-abbreviation>
      <score-instrument id="P1-I3">
        <instrument-name>Flute</instrument-name>
        </score-instrument>
      <midi-instrument id="P1-I3">
        <midi-channel>1</midi-channel>
        <midi-program>74</midi-program>
        <volume>78.7402</volume>
        <pan>0</pan>
        </midi-instrument>
      </score-part>
    </part-list>
  <part id="P1">
    <measure number="1" width="158.18">
      <print>
        <system-layout>
          <system-margins>
            <left-margin>65.78</left-margin>
            <right-margin>0.00</right-margin>
            </system-margins>
          <top-system-distance>70.00</top-system-distance>
          </system-layout>
        </print>
      <attributes>
        <divisions>1</divisions>
        <key>
          <fifths>1</fifths>
          <mode>major</mode>
          </key>
        <time>
          <beats>4</beats>
          <beat-type>4</beat-type>
          </time>
        <clef>
          <sign>G</sign>
          <line>2</line>
          </clef>
        </attributes>
      <note>
        <rest/>
        <duration>4</duration>
        <voice>1</voice>
        </note>
      </measure>
    <measure number="2" width="116.43">
      <attributes>
        <key>
        <fifths>2</fifths>
        <mode>major</mode>
        </key>
      </attributes>
    <note>
      <rest/>
      <duration>4</duration>
      <voice>1</voice>
      </note>
    </measure>
  <measure number="3" width="126.43">
    <attributes>
      <key>
        <fifths>3</fifths>
        <mode>major</mode>
        </key>
      </attributes>
    <note>
      <rest/>
      <duration>4</duration>
      <voice>1</voice>
      </note>
    </measure>
  <measure number="4" width="136.43">
    <attributes>
      <key>
        <fifths>4</fifths>
        <mode>major</mode>
        </key>
      </attributes>
    <note>
      <rest/>
      <duration>4</duration>
      <voice>1</voice>
      </note>
    </measure>
  <measure number="5" width="146.43">
    <attributes>
      <key>
        <fifths>5</fifths>
        <mode>major</mode>
        </key>
      </attributes>
    <note>
      <rest/>
      <duration>4</duration>
      <voice>1</voice>
      </note>
    </measure>
  <measure number="6" width="156.43">
    <attributes>
      <key>
        <fifths>6</fifths>
        <mode>major</mode>
        </key>
      </attributes>
    <note>
      <rest/>
      <duration>4</duration>
      <voice>1</voice>
      </note>
    </measure>
  <measure number="7" width="171.03">
    <attributes>
      <key>
        <fifths>7</fifths>
        <mode>major</mode>
        </key>
      </attributes>
    <note>
      <rest/>
      <duration>4</duration>
      <voice>1</voice>
      </note>
    <barline location="right">
      <bar-style>light-light</bar-style>
      </barline>
    </measure>
  <measure number="8" width="243.24">
    <print new-system="yes">
      <system-layout>
        <system-margins>
          <left-margin>42.50</left-margin>
          <right-margin>0.00</right-margin>
          </system-margins>
        <system-distance>92.50</system-distance>
        </system-layout>
      </print>
    <attributes>
      <key>
        <fifths>-7</fifths>
        <mode>major</mode>
        </key>
      </attributes>
    <note>
      <rest/>
      <duration>4</duration>
      <voice>1</voice>
      </note>
    </measure>
  <measure number="9" width="142.59">
    <attributes>
      <key>
        <fifths>-6</fifths>
        <mode>major</mode>
        </key>
      </attributes>
    <note>
      <rest/>
      <duration>4</duration>
      <voice>1</voice>
      </note>
    </measure>
  <measure number="10" width="132.59">
    <attributes>
      <key>
        <fifths>-5</fifths>
        <mode>major</mode>
        </key>
      </attributes>
    <note>
      <rest/>
      <duration>4</duration>
      <voice>1</voice>
      </note>
    </measure>
  <measure number="11" width="122.59">
    <attributes>
      <key>
        <fifths>-4</fifths>
        <mode>major</mode>
        </key>
      </attributes>
    <note>
      <rest/>
      <duration>4</duration>
```

```xml
        <voice>1</voice>
        </note>
      </measure>
    <measure number="12" width="112.59">
      <attributes>
        <key>
          <fifths>-3</fifths>
          <mode>major</mode>
          </key>
        </attributes>
      <note>
        <rest/>
        <duration>4</duration>
        <voice>1</voice>
        </note>
      </measure>
    <measure number="13" width="102.59">
      <attributes>
        <key>
          <fifths>-2</fifths>
          <mode>major</mode>
          </key>
        </attributes>
      <note>
        <rest/>
        <duration>4</duration>
        <voice>1</voice>
        </note>
      </measure>
    <measure number="14" width="92.59">
      <attributes>
        <key>
          <fifths>-1</fifths>
          <mode>major</mode>
          </key>
        </attributes>
      <note>
        <rest/>
        <duration>4</duration>
        <voice>1</voice>
        </note>
      </measure>
    <measure number="15" width="85.86">
      <attributes>
        <key>
          <fifths>0</fifths>
          <mode>major</mode>
          </key>
        </attributes>
      <note>
        <rest/>
        <duration>4</duration>
        <voice>1</voice>
        </note>
      <barline location="right">
        <bar-style>light-light</bar-style>
        </barline>
      </measure>
    </part>
  </score-partwise>
```

6 References

Audiveris (2015). *Open Music Scanner*. URL: `https://audiveris.kenai.com` (visited on 09/01/2015).

Avid (2015a). *Sibelius: the leading music composition and notation software*. URL: `http://www.sibelius.com/home/index_flash.html` (visited on 06/01/2015).

Bainbridge, David and Tim Bell (2015). *The Challenge of Optical Music Recognition*. URL: `http://www.eecs.harvard.edu/~cat/cs/omr/docs/bainbridge-bell-challenge-of-omr.pdf` (visited on 2001).

Feist, Jonathan (2012). *Sheet Music Cabinets: a rant*. URL: `http://jonathanfeist.berkleemusicblogs.com/2012/08/07/sheet-music-cabinets-a-rant/` (visited on 06/01/2015).

IMSLP (2015). *IMSLP/Petrucci Music Library*. URL: `http://imslp.org` (visited on 06/01/2015).

Lilypond (2015). *Lilypond - Music notation for everyone*. URL: `http://lilypond.org/index.html` (visited on 06/01/2015).

Make Music, Inc (2015). *MusicXML for Exchanging Digital Sheet Music*. URL: `http://www.musicxml.com` (visited on 06/01/2015).

MIT (2013). *Music21: A toolkit for computer-aided Musicology*. URL: `http://web.mit.edu/music21/` (visited on 01/13/2015).

MuseScore (2015a). *MuseScore Tour*. URL: `http://musescore.org/en/musescore-tour` (visited on 06/01/2015).

– (2015b). *Sheet Music Sharing*. URL: `http://musescore.com` (visited on 06/01/2015).

The Python Foundation (2015). *Python in Music*. URL: `https://wiki.python.org/moin/PythonInMusic` (visited on 06/01/2015).

Association, Major Orchestra Librarians' (2015). *The Orchestra Librarian*. URL: `http://mola-inc.org/page/Career` (visited on 04/07/2015).

Avid (2013). *Avid Scorch on the App Store on iTunes*. URL: `https://itunes.apple.com/app/avid-scorch/id436394592` (visited on 01/11/2015).

– (2015b). *PhotoScore Ultimate*. URL: `http://www.sibelius.com/products/photoscore/ultimate.html` (visited on 01/11/2015).

Deskew Technologies (2011). *Purchase*. URL: `http://www.deskew.com/products/scorecerer-desktop-pro.html` (visited on 01/11/2015).

ForScore, LLC (2014). *For Score*. URL: `https://itunes.apple.com/GB/app/id363738376?mt=8` (visited on 04/07/2015).

Fujisaki, T. et al. (1990). "Online Recognizer for Runon Handprinted Characters". In: *10th International Conference on Pattern Recognition* 1.

Good, Michael (2013). *Beyond PDF – Exchange and Publish Scores with MusicXML*. URL: `http://www.musicxml.com/wp-content/uploads/2013/04/MusicXML-Musikmesse-2013.pdf` (visited on 04/07/2015).

Kaufman, Walter (1967). *Musical Notations of the Orient*. Indiana University Press.

Land, Lois Rhea and Mary Ann Vaughan (1978). *Music in today's classroom: creating, listening, performing*. Harcourt Brace Jovanovich, Inc.

Make Music, Inc (2012). *Software - MusicXML*. URL: `http://www.musicxml.com/software/` (visited on 01/11/2015).

MusicReader (2015). *Music Reader*. URL: `http://www.musicreader.net/software.html` (visited on 04/07/2015).

Musicroom (2015). *Musicroom - Sheet Music for Musicians*. URL: `http://www.musicroom.com` (visited on 04/07/2015).

Newkirk, James W. and Alexei A. Vorontsov (2004). *Test-Driven Development in Microsoft .NET*. Microsoft Press.

Rastall, Richard (1982). *The Notation of Western Music: An Introduction*. St. Martin's Press.

Redmonk (2014). *Github Language Trends and the Fragmenting Landscape*. URL: `http://redmonk.com/dberkholz/2014/05/02/github-language-trends-and-the-fragmenting-landscape/` (visited on 01/11/2015).

SightRead Ltd. (2015). *http://www.sightread.co.uk/power-music.html*. URL: `http://www.sightread.co.uk/power-music.html` (visited on 01/11/2015).

Sullivan, Anne (2012). *Organize Your Sheet Music – What's Your System?* URL: `http://harpmastery.com/organize-your-sheet-music-whats-your-system/` (visited on 01/11/2015).

Taruskin, Richard (2005). *The Oxford History of Western Music*. Vol. 1. Oxford University Press.

The Mono Project (2015). *The Mono Development Project*. URL: `http://www.mono-project.com/docs/about-mono/languages/csharp/` (visited on 01/11/2015).

The Python Foundation (2012). *Python 2 or Python 3? - The Python Foundation*. URL: `https://wiki.python.org/moin/Python2orPython3` (visited on 01/11/2015).

The wxPython Team (2012). *PDF viewer - wxPython (phoenix) documentation*. URL: `http://wxpython.org/Phoenix/docs/html/lib.pdfviewer.html` (visited on 01/11/2015).

Warner, Sylvia Townsend (1929). *The Oxford History of Music, Introductory Volume*. Oxford University Press. Chap. 3.